aws

Guide du développeur

AWS Lambda

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS Lambda: Guide du développeur

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Les marques commerciales et la présentation commerciale d'Amazon ne peuvent pas être utilisées en relation avec un produit ou un service extérieur à Amazon, d'une manière susceptible d'entraîner une confusion chez les clients, ou d'une manière qui dénigre ou discrédite Amazon. Toutes les autres marques commerciales qui ne sont pas la propriété d'Amazon sont la propriété de leurs propriétaires respectifs, qui peuvent ou non être affiliés ou connectés à Amazon, ou sponsorisés par Amazon.

Table of Contents

Qu'est-ce que c'est AWS Lambda ? 1
Fonctionnement de Lambda 1
Fonctions principales 2
Informations connexes 3
Comment ça marche 4
Fonctions Lambda et gestionnaires de fonctions 5
Environnement d’exécution et exécutions Lambda 5
Événements et déclencheurs 6
Autorisations et rôles Lambda 7
Exécution du code 9
Création d’architectures guidées par les événements 26
Conception d’une application 39
Création de votre première fonction 47
Conditions préalables 47
Créer la fonction 49
Invoquer la fonction 56
Nettoyage 59
Étapes suivantes 60
Exemples d’applications et de modèles 62
Traitement des fichiers 62
Intégration de base de données 62
Tâches planifiées 63
Des flux de travail de longue durée 63
Ressources supplémentaires 63
Application de traitement de fichiers 64
Créer les fichiers de code source 65
Déployer l’application 68
Tester l'application 81
Étapes suivantes 88
Application de maintenance planifiée 89
Conditions préalables 90
Téléchargement des exemples de fichiers d’application 91
Création et remplissage de l’exemple de table DynamoDB 101
Création de l’application de maintenance planifiée 104

Test de l’application 109
Étapes suivantes 110
Création d'un système de traitement des commandes avec des fonctions Lambda Durable 110
Conditions préalables 110
Création des fichiers de code source 110
Déployer l'application 111
Vérification de l'application 112
Étapes suivantes 113
Outils de développement 115
Outils de développement locaux 115
Outils d’infrastructure en tant que code (IaC) 116
Outils GitHub Actions 117
Powertools pour AWS Lambda 117
Outils de gestion des flux de travail et des événements 118
Développement local 118
Principaux avantages du développement local 118
Prérequis 119
Authentification et contrôle d’accès 119
Déplacement de la console au développement local 122
Utilisation des fonctions localement 123
Convertir votre fonction en modèle AWS SAM et utiliser les outils IaC 126
Étapes suivantes 126
GitHub Actions 127
Exemple de flux de travail 127
Ressources supplémentaires 128
Infrastructure en tant que code (IaC) 128
Outils de l’IaC pour Lambda 128
Utilisation d’AWS SAM et d’Infrastructure Composer 130
En utilisant AWS CDK 142
Powertools pour AWS Lambda 152
Principaux avantages de Powertools pour AWS 152
Intégration de Powertools à AWS 153
Étapes suivantes 126
Flux de travail et événements 154
Orchestration axée sur le code avec fonctions durables 154
Orchestration de flux de travail avec Step Functions 156

Gestion des événements avec EventBridge et EventBridge planificateur 158
Fonctions durables Lambda 159
Principaux avantages 159
Comment ça marche 160
Quand utiliser des fonctions durables 161
Étapes suivantes 162
Concepts de base 162
Exécution durable 163
DurableContext 163
Étapes 164
États d'attente 165
Invocation d'autres fonctions 166
Configuration fonctionnelle durable 167
Création de fonctions durables 169
Conditions préalables 170
Créez une fonction durable 172
Invoquez la fonction durable 178
Nettoyage 180
Étapes suivantes 181
En utilisant AWS CLI 182
Utiliser l'infrastructure en tant que code 189
Exemples 195
Procédés de courte durée tolérants aux pannes 196
Procédés de longue durée 201
Invocations enchaînées entre les fonctions 208
Modèles avancés 212
Étapes suivantes 221
Sécurité et autorisations 221
Autorisations du rôle d’exécution 221
Chiffrement d'état 222
CloudTrail journalisation 223
Considérations relatives à l’accès intercompte 224
Fonctionnalités de sécurité Lambda héritées 224
SDK d'exécution durable 225
DurableContext 225
Ce que fait le SDK 226

Comment fonctionne le point de contrôle ... 226
Comportement de rediffusion ... 227
Opérations durables disponibles ... 228
Comment mesure-t-on la durabilité des opérations ... 235
Environnements d'exécution pris en charge ... 238
Invoquer des fonctions durables ... 242
Méthodes d'invocation ... 242
ARNs Exigence qualifiée ... 243
Comprendre le cycle de vie d'exécution ... 245
Invocation depuis le code de l'application ... 245
Invocations enchaînées ... 247
Mappages des sources d'événements ... 248
Nouvelles tentatives ... 255
Idempotence ... 263
Tester des fonctions durables ... 268
Tests locaux ... 269
Test cloud ... 271
Que faut-il tester ? ... 273
Stratégie de test ... 274
Défaillances de débogage ... 274
Surveillance des fonctions durables ... 275
CloudWatch métriques ... 275
EventBridge événements ... 277
AWS X-Ray traçage ... 279
Bonnes pratiques ... 280
Écrire du code déterministe ... 280
Conception axée sur l'idempuissance ... 284
Gérez efficacement l'état ... 286
Concevez des étapes efficaces ... 288
Utilisez efficacement les opérations d'attente ... 288
Considérations supplémentaires ... 289
Ressources supplémentaires ... 290
Instances gérées Lambda ... 291
Capacités clés ... 291
Quand utiliser les instances gérées Lambda ... 291
Comment ça marche ... 292

Modèle de simultanéité 292
Location et isolement 293
Comprendre les instances gérées 293
Tarification 293
En quoi les instances gérées Lambda diffèrent-elles du type de calcul Lambda (par défaut) 294
Étapes suivantes 295
Prise en main 295
Création d'une fonction d'instance gérée Lambda (console) 295
Création d'une fonction d'instance gérée Lambda (CLI)AWS 297
Concepts de base 303
Fournisseurs de capacité 304
Mise à l’échelle 308
Sécurité 312
Rôle d'opérateur 314
Environnement d’exécution 317
Publication de versions 319
Runtimes 320
Environnements d'exécution pris en charge 320
Considérations spécifiques à l'exécution 320
Étapes suivantes 321
environnement d'exécution Java 321
Exécution de Node.js 326
Runtime Python 331
environnement d'exécution .NET 334
Réseaux 338
Options de connectivité 339
Sous-réseau public avec passerelle Internet 339
Points de terminaison d’un VPC 340
Sous-réseau privé avec passerelle NAT 341
Choix d'une option de connectivité 341
Étapes suivantes 342
Contrôle 343
Métriques disponibles 343
Fréquence métrique et rétention 344
Afficher les métriques dans CloudWatch 344
Utiliser des métriques pour optimiser les performances 345

Étapes suivantes 345
Quotas 345
Quotas de demandes d'API Lambda 345
Quotas de ressources pour les instances gérées Lambda 346
Quotas de mappage de la source 346
Demande d’augmentation de quota 347
Étapes suivantes 348
Bonnes pratiques 348
Configuration du fournisseur de capacité 348
Sélection du type d'instance 348
Configuration de la fonction 349
Configuration de la mise à l'échelle 349
Sécurité 350
Optimisation des coûts 350
Surveillance et observabilité 351
Considérations spécifiques à l'exécution 351
Étapes suivantes 352
Résolution des problèmes 352
Problèmes de régulation et de mise à l'échelle 352
Problèmes de simultanéité 353
Problèmes de performance 355
Problèmes liés aux fournisseurs de capacité 356
Problèmes de surveillance et d'observabilité 357
Aide supplémentaire 360
Étapes suivantes 360
Environnements d’exécution (runtimes) Lambda 361
Environnements d'exécution pris en charge 362
Nouvelles versions d'exécution 365
politique d’obsolescence de l’exécution 366
Modèle de responsabilité partagée 367
Environnement d’exécution utilisé après la date d’obsolescence 369
Réception de notifications d'obsolescence lors de l'exécution 371
Environnements d’exécution obsolètes 372
Mises à jour de version de l’environnement d’exécution 376
Rétrocompatibilité 377
Modes de mise à jour de l’environnement d’exécution 379

Déploiement des versions de l'environnement d'exécution en deux phases 380
Configuration de la gestion de l'environnement d'exécution 381
Rétablissement d'une version de l'environnement d'exécution 383
Mises à jour de version de l'environnement d'exécution 385
Modèle de responsabilité partagée 387
Autorisations 389
Obtenir des données sur les fonctions par environnement d'exécution 390
Liste des versions de fonctions qui utilisent un environnement d'exécution particulier 390
Identification des fonctions les plus fréquemment et les plus récemment invoquées 393
Modifications de l'environnement d'exécution 397
Variables d'environnement spécifiques à un langage 397
Scripts encapsuleurs 397
API de runtime 401
Invocation suivante 402
Réponse d'invocation 403
Erreur d'initialisation 403
Erreur d'invocation 405
Exécutions uniquement basées sur le système d'exploitation 408
Création d'un environnement d'exécution personnalisé 409
Environnement d'exécution personnalisé 415
Référentiels open source 423
Clients d'interface d'exécution 424
Bibliothèques d'événements 424
Images de base de conteneur 425
Outils de développement 425
Exemples de projets 427
Configuration des fonctions 428
Archives de fichiers .zip 430
Création de la fonction 431
Utilisation de l'éditeur de code de la console 432
Mise à jour du code de fonction 432
Modification de l'environnement d'exécution 433
Modification de l'architecture 434
Utilisation de l'API Lambda 434
Téléchargement du code de votre fonction 435
CloudFormation 435

Chiffrement ... 436
Images de conteneurs ... 443
Exigences ... 445
Utilisation d'une image AWS de base ... 445
Utilisation d'une image de base AWS uniquement pour le système d'exploitation ... 446
Utilisation d'une image non AWS basique ... 447
Clients d’interface d’exécution ... 448
Autorisations Amazon ECR ... 448
Cycle de vie des fonctions ... 451
Mémoire ... 453
Cas d’augmentation de la mémoire ... 453
Utilisation de la console ... 454
À l'aide du AWS CLI ... 454
En utilisant AWS SAM ... 455
Acceptation des recommandations relatives à la mémoire d’une fonction (console) ... 455
Stockage éphémère ... 456
Cas d’utilisation ... 456
Utilisation de la console ... 457
À l'aide du AWS CLI ... 457
En utilisant AWS SAM ... 458
Jeux d'instructions (ARM/x86) ... 459
Avantages liés à l'utilisation de l'architecture arm64 ... 459
Exigences pour la migration vers l'architecture arm64 ... 460
Compatibilité des codes de fonction avec l'architecture arm64 ... 460
Comment migrer vers l'architecture arm64 ... 461
Configuration de l'architecture de l'ensemble des instructions ... 462
Timeout ... 463
Cas d’augmentation du délai d’expiration ... 463
Utilisation de la console ... 464
En utilisant le AWS CLI ... 464
En utilisant AWS SAM ... 464
Configurer des fonctions durables ... 465
Permettre une exécution durable ... 465
Autorisations IAM pour des fonctions durables ... 466
Bonnes pratiques en matière de configuration ... 468
Variables d’environnement ... 470

Créer des variables d’environnement 470
Exemple de scénario pour les variables d’environnement 474
Récupérer les variables d’environnement 476
Variables d’environnement d’exécution définies 477
Sécurisation des variables d’environnement 480
Attachement de fonctions à un VPC 484
Autorisations IAM requises 484
Associer des fonctions Lambda à un Amazon VPC dans votre Compte AWS 486
Accès à Internet en cas d’attachement à un VPC 490
IPv6 soutien 490
Bonnes pratiques d'utilisation de Lambda avec Amazon VPCs 491
Comprendre les interfaces réseau élastiques Hyperplane () ENIs 493
Utilisation des clés de condition IAM pour les paramètres du VPC 494
Didacticiels de VPC 499
Attachement de fonctions aux ressources d’un autre compte 501
Conditions préalables 501
Création d’un Amazon VPC dans le compte de votre fonction 502
Octroi d’autorisations VPC au rôle d’exécution de votre fonction 502
Création d’une requête de connexion d’appairage de VPC 503
Préparation du compte de votre ressource 504
Mise à jour de la configuration VPC dans le compte de votre fonction 505
Test de votre fonction 507
Accès Internet pour les fonctions VPC 508
Réseaux entrants 534
Considérations relatives aux points de terminaison d'interface Lambda 534
Création d'un point de terminaison d'interface pour Lambda 535
Création d'une stratégie de point de terminaison d'interface pour Lambda 537
Système de fichiers 539
Régions prises en charge 539
Rôle d'exécution et autorisations utilisateur 539
Configuration d'un système de fichiers et d'un point d'accès 540
Connexion à un système de fichiers (console) 541
Alias 543
Utilisation des alias 545
Alias pondéré 546
Versions 551

Création de versions de fonction 552
Utilisation des versions 553
Octroi d'autorisations 554
Étiquettes 556
Autorisations requises pour l'utilisation des balises 556
Utilisation des étiquettes avec la console 557
Utilisation de balises avec AWS CLI 558
Streaming des réponses 561
Limitation de la bande passante pour la diffusion des réponses 562
Compatibilité du VPC avec la diffusion en continu de réponses 562
Écriture de fonctions 563
Invocation de fonctions 565
Tutoriel : création d'une fonction de streaming de réponses avec une URL de la fonction 567
Invocation de fonctions 572
Invocation d'une fonction de manière synchrone 574
appel asynchrone 578
Gestion des erreurs 579
Configuration 580
Conservation des enregistrements 582
Invocation 593
Limites d'invocation synchrone 593
Invocation asynchrone pour les flux de travail de longue durée 593
Gestion de l'exécution APIs 594
Mappages des sources d'événements 596
Mappages des source d'événement et déclencheurs 597
Comportement de traitement par lots 597
Mode alloué 601
API de mappage de la source d'événement 603
Balises de mappage des sources d'événements 604
Filtrage des événements 609
Présentation des principes de base du filtrage d'événements 610
Traitement des enregistrements ne répondant pas aux critères de filtrage 612
Syntaxe des règles de filtrage 613
Attacher des critères de filtre à un mappage de sources d'événements (console) 615
Attacher des critères de filtre à un mappage de sources d'événements (AWS CLI) 616
Attacher des critères de filtre à un mappage de sources d'événements (AWS SAM) 617

Chiffrement des critères de filtre 618
Utilisation de filtres avec différents Services AWS 624
Test dans la console 626
Invocation de fonctions avec des événements de test 626
Création d'événements de test privés 627
Création d'événements de test partageables 627
Suppression des schémas d'événements de test partageables 629
États des fonctions 630
États de fonction lors des mises à jour 631
Relances 633
Détection de boucle récursive 635
Comprendre la détection de boucles récursives 635
Soutenu Services AWS et SDKs 637
Notifications de boucles récursives 640
Répondre aux notifications de détection de boucles récursives 641
Octroi d'exécution d'une fonction Lambda dans une boucle récursive 642
Régions prises en charge pour la détection des boucles récursives de Lambda 644
Fonction URLs 645
Création d'une URL de fonction (console) 646
Création d'URL de fonction (AWS CLI) 648
Ajouter une URL de fonction à un CloudFormation modèle 649
Partage des ressources cross-origin (CORS) 651
Fonction d'étranglement URLs 652
Fonction de désactivation URLs 652
Fonction de suppression URLs 653
Contrôle d'accès 654
Invocation d'URL de fonctions 665
URL de fonctions de surveillance 678
URLs Function et Amazon API Gateway 679
Didacticiel : création d'un point de terminaison de webhook 685
Mise à l'échelle de fonction 699
Comprendre et visualiser la simultanéité 699
Calcul de la simultanéité d'une fonction 704
Présentation de la simultanéité réservée et de la simultanéité allouée 706
Simultanéité réservée 706
Simultanéité provisionnée 709

Comment Lambda alloue la simultanéité provisionnée 714
Comparaison de la simultanéité réservée et de la simultanéité provisionnée 714
Présentation de la simultanéité et des requêtes par seconde 715
Quotas de simultanéité 717
Configuration de la simultanéité réservée 720
Configuration de la simultanéité réservée 721
Estimation précise de la simultanéité réservée requise pour une fonction 723
Configuration de la simultanéité provisionnée 725
Configuration de la simultanéité provisionnée 726
Estimation précise de la simultanéité provisionnée requise pour une fonction 728
Optimisation du code de fonction lors de l'utilisation de la simultanéité provisionnée 729
Utilisation de variables d'environnement pour visualiser et contrôler le comportement de simultanéité provisionnée 730
Comprendre le comportement de journalisation et de facturation avec la simultanéité provisionnée 731
Utilisation d'Application Auto Scaling pour automatiser la gestion de la simultanéité provisionnée 731
Comportement de mise à l'échelle. 736
Taux de mise à l'échelle de la simultanéité 736
Surveillance de la simultanéité 738
Métriques de simultanéité générales 738
Métriques de simultanéité provisionnée 738
Travailler avec la métrique `ClaimedAccountConcurrency` 741
Création avec Node.js 745
Versions du SDK incluses dans l'environnement d'exécution 747
Utilisation de keep-alive 747
Chargement des certificats CA 747
Fonctionnalités expérimentales de Node.js 748
Handler (Gestionnaire) 750
Configuration de votre projet 750
Exemple de fonction 751
Modules CommonJS et ES 753
Initialisation de Node.js 755
Convention de nommage du gestionnaire 755
Objet d'événement d'entrée 756
Modèles de gestionnaire valides 756

Utilisation du SDK pour JavaScript ... 758
Accès aux variables d'environnement ... 760
Utilisation de l'état global ... 760
Bonnes pratiques ... 760
Déploiement d'archives de fichiers .zip ... 763
Dépendances d'exécution dans Node.js ... 763
Création d'un package de déploiement .zip sans dépendances ... 764
Création d'un package de déploiement .zip avec dépendances ... 764
Création d'une couche Node.js pour vos dépendances ... 766
Chemin de recherche des dépendances et bibliothèques incluses dans l'exécution ... 767
Création et mise à jour de fonctions Lambda Node.js à l'aide de fichiers .zip ... 768
Déploiement d'images de conteneur ... 776
AWS images de base pour Node.js ... 777
Utilisation d'une image AWS de base ... 778
Utilisation d'une image non AWS basique ... 784
Layers ... 795
Empaqueter le contenu de votre couche ... 795
Créer la couche dans Lambda ... 800
Ajouter la couche à votre fonction ... 801
Exemple d'application ... 802
Contexte ... 803
Logging ... 805
Création d'une fonction qui renvoie des journaux ... 805
Utilisation des contrôles de journalisation avancés de Lambda avec Node.js ... 807
Affichage des journaux dans la console Lambda ... 814
Afficher les journaux dans la CloudWatch console ... 814
Afficher les journaux à l'aide de AWS Command Line Interface (AWS CLI) ... 815
Suppression de journaux ... 818
Tracing ... 819
Utilisation d'ADOT pour instrumenter vos fonctions Node.js ... 820
Utilisation du kit SDK X-Ray pour instrumenter vos fonctions Node.js ... 820
Activation du suivi avec la console Lambda ... 821
Activation du suivi avec l'API Lambda ... 822
Activation du traçage avec CloudFormation ... 822
Interprétation d'un suivi X-Ray ... 823
Stockage des dépendances d'exécution dans une couche (kit SDK X-Ray) ... 826

Construire avec TypeScript 828
Environnement de développement 829
Définitions de type pour Lambda 830
Handler (Gestionnaire) 832
Configuration de votre projet 832
Exemple de fonction 833
Modules CommonJS et ES 836
Initialisation de Node.js 836
Convention de nommage du gestionnaire 837
Objet d'événement d'entrée 837
Modèles de gestionnaire valides 838
Utilisation du SDK pour JavaScript 841
Accès aux variables d'environnement 842
Utilisation de l'état global 842
Bonnes pratiques 842
Déployez des archives de fichiers .zip 845
En utilisant AWS SAM 845
À l'aide du AWS CDK 847
Utilisation de AWS CLI et esbuild 850
Déployez les images du conteneur 853
Utilisation d'une image de base Node.js pour créer et conditionner le code de fonction TypeScript 853
Contexte 861
Logging 864
Outils et bibliothèques 864
Utilisation de Powertools pour AWS Lambda (TypeScript) et AWS SAM pour la journalisation structurée 865
Utilisation de Powertools pour AWS Lambda (TypeScript) et AWS CDK pour la journalisation structurée 868
Affichage des journaux dans la console Lambda 871
Afficher les journaux dans la CloudWatch console 872
Tracing 873
Utilisation de Powertools pour AWS Lambda (TypeScript) et AWS SAM pour le traçage 874
Utilisation de Powertools pour AWS Lambda (TypeScript) et AWS CDK pour le traçage 876
Interprétation d'un suivi X-Ray 880
Création avec Python 881

Versions du SDK incluses dans l'environnement d'exécution 882
Fonctionnalités Python désactivées 883
Format de la réponse 883
Arrêt progressif pour les extensions 884
Handler (Gestionnaire) 885
Exemple de fonction Lambda Python 885
Convention de nommage du gestionnaire 887
Utilisation de l'objet d'événement Lambda 888
Accès et utilisation de l'objet de contexte Lambda 889
Signatures de gestionnaire valides pour les gestionnaires Python 890
Renvoi d'une valeur 890
Utilisation d'AWS SDK pour Python (Boto3) dans votre gestionnaire 891
Accès aux variables d'environnement 892
Pratiques exemplaires en matière de code pour les fonctions Lambda Python 893
Déploiement d'archives de fichiers .zip 895
Dépendances d'exécution dans Python 895
Création d'un package de déploiement .zip sans dépendances 897
Création d'un package de déploiement .zip avec dépendances 897
Chemin de recherche des dépendances et bibliothèques incluses dans l'exécution 900
Utilisation des dossiers __pycache__ 901
Création de packages de déploiement .zip avec des bibliothèques natives 901
Création et mise à jour de fonctions Lambda Python à l'aide de fichiers .zip 903
Déploiement d'images de conteneur 911
AWS images de base pour Python 912
Utilisation d'une image AWS de base 914
Utilisation d'une image non AWS basique 920
Layers 930
Empaqueter le contenu de votre couche 930
Créer la couche dans Lambda 800
Ajouter la couche à votre fonction 936
Exemple d'application 937
Contexte 938
Logging 940
Impression dans le journal 940
Utilisation d'une bibliothèque de journalisation 941
Utilisation des contrôles de journalisation avancés de Lambda avec Python 943

Affichage des journaux dans la console Lambda 948
Afficher les journaux dans CloudWatch la console 948
Afficher les journaux avec AWS CLI 949
Suppression de journaux 952
Outils et bibliothèques 952
Utilisation de Powertools pour AWS Lambda (Python) et AWS SAM pour la journalisation structurée 953
Utilisation de Powertools pour AWS Lambda (Python) et AWS CDK pour la journalisation structurée 957
Test 964
Test de vos applications sans serveur 965
Tracing 967
Utilisation de Powertools pour AWS Lambda (Python) et d'AWS SAM pour le traçage 968
Utilisation de Powertools pour AWS Lambda (Python) et de AWS CDK pour le traçage 971
Utilisation d'ADOT pour instrumenter vos fonctions python 976
Utilisation du kit SDK X-Ray pour instrumenter vos fonctions Python 976
Activation du suivi avec la console Lambda 977
Activation du suivi avec l'API Lambda 977
Activation du suivi avec CloudFormation 978
Interprétation d'un suivi X-Ray 978
Stockage des dépendances d'exécution dans une couche (kit SDK X-Ray) 981
Création avec Ruby 983
Versions du SDK incluses dans l'environnement d'exécution 985
Activer un autre JIT Ruby (YJIT) 985
Handler (Gestionnaire) 986
Notions de base sur le gestionnaire Ruby 986
Pratiques exemplaires en matière de code pour les fonctions Lambda Ruby 987
Déploiement d'archives de fichiers .zip 990
Dépendances dans Ruby 991
Création d'un package de déploiement .zip sans dépendances 991
Création d'un package de déploiement .zip avec dépendances 991
Création d'une couche Ruby pour vos dépendances 993
Création de packages de déploiement .zip avec des bibliothèques natives 993
Création et mise à jour de fonctions Lambda Ruby à l'aide de fichiers .zip 995
Déploiement d'images de conteneur 1003
AWS images de base pour Ruby 1004

Utilisation d'une image AWS de base ... 1004
Utilisation d'une image non AWS basique ... 1011
Couches ... 1022
Empaqueter le contenu de votre couche ... 1022
Créer la couche dans Lambda ... 800
Utiliser des gems provenant de couches dans une fonction ... 1030
Ajouter la couche à votre fonction ... 1030
Exemple d’application ... 1031
Contexte ... 1032
Logging ... 1033
Création d’une fonction qui renvoie des journaux ... 1033
Affichage des journaux dans la console Lambda ... 1035
Afficher les journaux dans la CloudWatch console ... 1035
Afficher les journaux à l'aide de AWS Command Line Interface (AWS CLI) ... 1035
Suppression de journaux ... 1038
Utilisation de la bibliothèque Ruby logger ... 1039
Tracing ... 1040
Activation du suivi actif avec l’API Lambda ... 1046
Activation du suivi actif avec CloudFormation ... 1046
Stockage des dépendances d’exécution dans une couche ... 1047
Création avec Java ... 1048
Handler (Gestionnaire) ... 1052
Configuration de votre projet de gestionnaire Java ... 1052
Exemple de code de fonction Lambda Java ... 1053
Définitions de classe valides pour les gestionnaires Java ... 1058
Convention de nommage du gestionnaire ... 1060
Définition et accès à l’objet d’événement d’entrée ... 1060
Accès et utilisation de l’objet de contexte Lambda ... 1061
Utilisation du SDK AWS pour Java v2 dans votre gestionnaire ... 1062
Accès aux variables d’environnement ... 1064
Utilisation de l'état global ... 1064
Pratiques exemplaires en matière de code pour les fonctions Lambda Java ... 1064
Déploiement d’archives de fichiers .zip ... 1067
Conditions préalables ... 1068
Outils et bibliothèques ... 1068
Création d’un package de déploiement avec Gradle ... 1069

Utilisation des couches pour les dépendances 1070
Création d'un package de déploiement avec Maven 1070
Chargement d'un package de déploiement avec la console Lambda 1073
Téléchargement d'un package de déploiement à l'aide du AWS CLI 1074
Téléchargement d'un package de déploiement avec AWS SAM 1076
Déploiement d'images de conteneur 1079
AWS images de base pour Java 1080
Utilisation d'une image AWS de base 1081
Utilisation d'une image non AWS basique 1091
Layers 1102
Empaqueter le contenu de votre couche 1102
Créer la couche dans Lambda 800
Ajouter la couche à votre fonction 1106
Sérialisation personnalisée 1107
Cas d'utilisation de la sérialisation personnalisée 1107
Implémentation de sérialisation personnalisée 1108
Test de la sérialisation personnalisée 1109
Comportement de démarrage personnalisé 1110
Présentation de la variable d'environnement `JAVA_TOOL_OPTIONS` 1110
Patch Log4j pour Log4Shell 1113
Ahead-of-Time (AOT) et caches CDS 1113
Contexte 1115
Contexte dans des exemples d'applications 1117
Logging 1119
Création d'une fonction qui renvoie des journaux 1119
Utilisation des contrôles de journalisation avancés de Lambda avec Java 1121
Implémentation de la journalisation avancée avec Log4j2 et J SLF4 1125
Outils et bibliothèques 1128
Utilisation de Powertools pour AWS Lambda (Java) et AWS SAM pour la journalisation structurée 1129
Affichage des journaux dans la console Lambda 1133
Afficher les journaux dans la CloudWatch console 1134
Afficher les journaux à l'aide de AWS Command Line Interface (AWS CLI) 1134
Suppression de journaux 1137
Exemple de code de journalisation 1137
Tracing 1139

Utilisation de Powertools pour AWS Lambda (Java) et AWS SAM pour le traçage 1140
Utilisation de Powertools pour AWS Lambda (Java) et AWS CDK pour le traçage 1142
Utilisation d'ADOT pour instrumenter vos fonctions Java 1154
Utilisation du kit SDK X-Ray pour instrumenter vos fonctions Java 1155
Activation du suivi avec la console Lambda 1155
Activation du suivi avec l'API Lambda 1156
Activation du traçage avec CloudFormation 1156
Interprétation d'un suivi X-Ray 1157
Stockage des dépendances d'exécution dans une couche (kit SDK X-Ray) 1160
Suivi de X-Ray dans des exemples d'applications (kit SDK X-Ray) 1161
Exemples d'application 1162
Création avec Go 1164
Prise en charge de l'exécution Go 1164
Outils et bibliothèques 1165
Handler (Gestionnaire) 1167
Configuration de votre projet de gestionnaire Go 1167
Exemple de fonction Lambda Go 1168
Convention de nommage du gestionnaire 1171
Définition et accès à l'objet d'événement d'entrée 1171
Accès et utilisation de l'objet de contexte Lambda 1172
Signatures de gestionnaire valides pour les gestionnaires Go 1173
Utilisation de l'AWS SDK pour Go v2 dans votre gestionnaire 1174
Accès aux variables d'environnement 1175
Utilisation de l'état global 1176
Pratiques exemplaires en matière de code pour les fonctions Lambda Go 1176
Contexte 1178
Variables, méthodes et propriétés prises en charge par l'objet de contexte 1178
Accès aux informations du contexte d'appel 1179
Utilisation du contexte dans les initialisations et les appels du client du kit SDK AWS 1181
Déploiement d'archives de fichiers .zip 1182
Création d'un fichier .zip sur macOS et Linux 1182
Création d'un fichier .zip sous Windows 1184
Création et mise à jour de fonctions Lambda Go à l'aide de fichiers .zip 1187
Déploiement d'images de conteneur 1194
AWS images de base pour le déploiement des fonctions Go 1194
Client d'interface d'environnement d'exécution Go 1195

Utilisation d'une image de base AWS uniquement pour le système d'exploitation 1195
Utilisation d'une image non AWS basique 1203
Couches 1212
Logging 1213
Création d’une fonction qui renvoie des journaux 1213
Affichage des journaux dans la console Lambda 1215
Afficher les journaux dans la CloudWatch console 1215
Afficher les journaux à l'aide de AWS Command Line Interface (AWS CLI) 1216
Suppression de journaux 1219
Tracing 1220
Utilisation d'ADOT pour instrumenter vos fonctions Go 1221
Utilisation du kit SDK X-Ray pour instrumenter vos fonctions Go 1221
Activation du suivi avec la console Lambda 1221
Activation du suivi avec l’API Lambda 1222
Activation du suivi avec CloudFormation 1222
Interprétation d’un suivi X-Ray 1223
Création avec C# 1227
Environnement de développement 1227
Installation des modèles de projet .NET 1227
Installation et mise à jour des outils CLI 1228
Handler (Gestionnaire) 1229
Configuration de votre projet de gestionnaire C# 1230
Exemple de fonction Lambda C# 1231
Gestionnaires de bibliothèques de classes 1234
Gestionnaires d'assemblages exécutables 1235
Signatures de gestionnaire valides pour les fonctions C# 1236
Convention de nommage du gestionnaire 1237
Sérialisation dans les fonctions Lambda C# 1237
Fonctions basées sur des fichiers 1240
Accès et utilisation de l’objet de contexte Lambda 1241
Utilisation de la SDK pour .NET v3 dans votre gestionnaire 1242
Accès aux variables d’environnement 1243
Utilisation de l'état global 1243
Simplifiez le code de la fonction à l'aide du cadre d'annotations Lambda 1244
Pratiques exemplaires de codage pour les fonctions Lambda C# 1245
Package de déploiement 1248

CLI .NET Lambda Global 1249
AWS SAM 1255
AWS CDK 1259
ASP.NET 1262
Couches 1268
Déploiement d’images de conteneur 1269
AWS images de base pour .NET 1270
Utilisation d'une image AWS de base 1270
Utilisation d'une image non AWS basique 1273
Compilation anticipée native 1277
Le fichier d’exécution Lambda 1277
Prérequis 1278
Prise en main 1278
Sérialisation 1282
Réduction 1282
Résolution des problèmes 1283
Contexte 1284
Logging 1286
Création d’une fonction qui renvoie des journaux 1286
Utilisation des contrôles de journalisation avancés de Lambda avec .NET 1287
Outils et bibliothèques 1295
Utilisation de Powertools pour AWS Lambda (.NET) et AWS SAM pour la journalisation structurée 1296
Affichage des journaux dans la console Lambda 1299
Afficher les journaux dans la CloudWatch console 1299
Afficher les journaux à l'aide de AWS Command Line Interface (AWS CLI) 1299
Suppression de journaux 1303
Tracing 1304
Utilisation de Powertools pour AWS Lambda (.NET) et de AWS SAM pour le traçage 1305
Utilisation du kit SDK X-Ray pour instrumenter vos fonctions .NET 1308
Activation du suivi avec la console Lambda 1309
Activation du suivi avec l’API Lambda 1310
Activation du suivi avec CloudFormation 1310
Interprétation d’un suivi X-Ray 1311
Test 1315
Test de vos applications sans serveur 1316

Construire avec PowerShell 1320
Environnement de développement 1322
Package de déploiement 1323
Création d'une fonction Lambda 1323
Handler (Gestionnaire) 1326
Renvoi de données 1327
Contexte 1328
Logging 1329
Création d’une fonction qui renvoie des journaux 1329
Affichage des journaux dans la console Lambda 1331
Afficher les journaux dans la CloudWatch console 1331
Afficher les journaux à l'aide de AWS Command Line Interface (AWS CLI) 1332
Suppression de journaux 1335
Création avec Rust 1336
Handler (Gestionnaire) 1338
Configuration de votre projet de gestionnaire Rust 1338
Exemple de fonction Lambda Rust 1339
Définitions de classe valides pour les gestionnaires Rust 1342
Convention de nommage du gestionnaire 1343
Définition et accès à l’objet d’événement d’entrée 1344
Accès et utilisation de l’objet de contexte Lambda 1345
En utilisant le AWS SDK pour Rust dans votre gestionnaire 1345
Accès aux variables d’environnement 1346
Utilisation de l’état partagé 1347
Pratiques exemplaires en matière de code pour les fonctions Lambda Rust 1347
Contexte 1350
Accès aux informations du contexte d’appel 1350
Événements HTTP 1352
Déploiement d’archives de fichiers .zip 1355
Conditions préalables 1355
Création de la fonction 1355
Déploiement de la fonction 1356
Invoquer la fonction 1358
Couches 1359
Logging 1360
Création d’une fonction qui écrit des journaux 1360

Implémentation de la journalisation avancée avec la caisse Tracing 1360
Bonnes pratiques 1363
Code de fonction 1363
Configuration de la fonction 1365
Capacité de mise à l'échelle de la fonction 1366
Métriques et alarmes 1367
Utilisation des flux 1368
Bonnes pratiques de sécurité 1369
Test de fonctions sans serveur 1371
Résultats commerciaux ciblés 1372
Que faut-il tester ? 1372
Comment tester le sans serveur 1373
Techniques de test 1374
Tests dans le cloud 1375
Tester avec des simulations 1378
Tester localement en utilisant AWS SAM CLI 1379
Tester avec émulation 1380
Bonnes pratiques 1381
Prioriser les tests dans le cloud 1381
Structurer votre code pour le rendre testable 1381
Accélérer les boucles de rétroaction du développement 1382
Se concentrer sur les tests d'intégration 1382
Créer des environnements de test isolés 1383
Utiliser des simulations pour isoler la logique métier 1384
Utiliser les émulateurs avec parcimonie 1385
Défis liés aux tests locaux 1385
Exemple : la fonction Lambda crée un compartiment S3 1386
Exemple : la fonction Lambda traite les messages d'une file d'attente Amazon SQS 1386
FAQ 1387
Prochaines étapes et ressources 1388
Lambda SnapStart 1390
Cas d'utilisation 1391
Fonctions prises en charge et limitations 1391
Régions prises en charge 1392
Considérations de compatibilité 1392
Tarification 1393

Activation SnapStart 1395
Activation SnapStart (console) 1395
Activation SnapStart (AWS CLI) 1396
Activation SnapStart (API) 1398
États des fonctions 1399
Mise à jour d'un instantané 1399
Utilisation SnapStart avec AWS SDKs 1400
Utilisation SnapStart avec CloudFormationAWS SAM, et AWS CDK 1400
Suppression d'instantanés 1400
Gestion de l'unicité 1402
Évitez de sauvegarder un état 1402
Utiliser des CSPRNG 1404
Outil d'analyse (Java) 1407
Hooks d'exécution 1408
Java 1408
Python 1412
.NET 1414
Surveillance 1417
CloudWatch Logs 1417
AWS X-Ray 1418
API de télémétrie 1419
Métriques d'API Gateway et d'URL de la fonction 1419
Modèle de sécurité 1420
Bonnes pratiques 1421
Personnalisation de performances 1421
Bonnes pratiques concernant le réseau 1425
Résolution des problèmes 1427
SnapStartNotReadyException 1427
SnapStartTimeoutException 1427
Erreur interne du service 500 1428
401 Accès non autorisé 1428
UnknownHostException (Java) 1428
Échecs de création d'instantanés 1429
Latence de création d'instantanés 1429
Isolement des locataires 1431
Quand utiliser le mode d'isolation des locataires 1432

Fonctions prises en charge et limitations 1432
Soutenu Régions AWS 1432
Considérations 1433
Tarification 1433
Mode d'isolation 1433
Permettre l'isolement des locataires 1435
Console 1435
AWS CLI 1435
API 1436
CloudFormation 1436
Invocation de fonctions avec isolation des locataires 1438
AWS CLI 1438
API 1438
API Gateway 1439
Kit SDK 1440
Accès en code 1443
Accès 1443
Schémas d'utilisation 1444
Contrôle 1445
Contrôle 1447
Comprendre la journalisation en mode isolé pour les locataires 1447
Résolution des problèmes 1449
InvalidParameterValueException 1449
TooManyRequestsException 1449
Intégration d’autres services 1450
Création d’un déclencheur 1450
Liste des services 1451
Apache Kafka 1454
MSK 1456
Self-managed Apache Kafka 1507
Mise à l’échelle des sondeurs d’événements 1527
Positions des interrogations et des flux 1535
ID de groupe de consommateurs 1536
Filtrage des événements 1537
Registres de schémas avec sources d’événements 1543
Apache Kafka à faible latence 1572

Réessayer les configurations 1574
Conserver les appels ayant échoué 1580
Destination de Kafka en cas d'échec 1588
Résolution des problèmes 1593
API Gateway 1596
Choix d’un type d’API 1597
Ajout d’un point de terminaison public à votre fonction Lambda 1598
Intégration de proxy 1599
Format des événements 1599
Format de la réponse 1599
Permissions 1600
Exemple d’application 1603
Le gestionnaire d'événements de Powertools pour Lambda AWS 1603
didacticiel 1604
Erreurs 1621
API Gateway et fonction URLs 1623
Compositeur d'infrastructure 1628
Exportation d’une fonction Lambda vers Infrastructure Composer 1628
Autres ressources 1631
CloudFormation 1632
Amazon DocumentDB 1635
Exemple d’événement Amazon DocumentDB 1636
Conditions préalables et autorisations 1637
Configurer la sécurité réseau 1639
Création d’un mappage des sources d’événements Amazon DocumentDB (console) 1642
Création d’un mappage des sources d’événements Amazon DocumentDB (kit SDK ou CLI) 1644
Positions de départ des interrogations et des flux 1647
Surveillance de votre source d’événements Amazon DocumentDB 1647
didacticiel 1648
DynamoDB 1675
Flux d’interrogation et de mise en lots 1676
Positions de départ des interrogations et des flux 1677
Lecteurs simultanés 1677
Exemple d’évènement 1677
Créer un mappage 1679

Erreurs d'éléments par lots 1682
Gestion des erreurs 1699
Traitement avec état 1706
Paramètres 1711
Filtrage des événements 1714
didacticiel 1723
EC2 1741
Octroi d'autorisations à EventBridge (CloudWatch Events) 1742
Elastic Load Balancing (Application Load Balancer) 1743
Gestionnaire d'événements de Powertools pour Lambda AWS 1745
Invoquer à l'aide du planificateur EventBridge 1747
Configurer le rôle d'exécution 1747
Créer une planification 1747
Ressources connexes 1752
IoT 1753
Kinesis Data Streams 1755
Flux d'interrogation et de mise en lots 1755
Exemple d'évènement 1758
Créer un mappage 1759
Erreurs d'éléments par lots 1764
Gestion des erreurs 1783
Traitement avec état 1790
Paramètres 1794
Filtrage des événements 1797
didacticiel 1802
Kubernetes 1819
AWS Controllers for Kubernetes (ACK) 1819
Crossplane 1820
MQ 1822
Comprendre le groupe de consommateurs Lambda pour Amazon MQ 1824
Configurer une source d'événements 1828
Paramètres 1835
Filtrage des événements 1836
Dépannage 1843
RDS 1845
Configuration de votre fonction pour qu'elle fonctionne avec les ressources RDS 1845

Connexion à une base de données Amazon RDS dans une fonction Lambda 1852
Traitement des notifications d'événements provenant d'Amazon RDS 1871
Tutoriel complet Lambda et Amazon RDS 1872
Amazon RDS et DynamoDB 1872
S3 1877
Didacticiel : Utiliser un déclencheur S3 1879
Didacticiel : Utilisation d'un déclencheur Amazon S3 pour créer des images miniatures 1904
Secrets Manager 1933
Choix d'une approche 1933
Quand utiliser Secrets Manager 1934
Utilisation des AWS paramètres et des secrets de l'extension Lambda 1934
Utilisation de l'utilitaire de paramètres de Powertools pour AWS Lambda 1945
SQS 1952
Comprendre le comportement d'interrogation et de traitement par lots pour les mappages des sources d'événements Amazon SQS 1953
Utilisation du mode provisionné avec les mappages de sources d'événements Amazon SQS 1954
Configuration du mode provisionné pour le mappage des sources d'événements Amazon SQS 1956
Exemple d'événement de message de file d'attente standard 1957
Exemple d'événement de message de file d'attente FIFO 1959
Création d'un mappage 1960
Comportement de mise à l'échelle. 1965
Gestion des erreurs 1968
Parameters 1983
Filtrage des événements 1985
didacticiel 1990
Tutoriel entre comptes SQS 2008
Step Functions 2015
Quand utiliser Step Functions 2015
Quand ne pas utiliser Step Functions 2022
Lot S3 2024
Appel de fonctions Lambda à partir d'opérations par lot Amazon S3 2025
SNS 2027
Utilitaire d'idempotence de Powertools pour AWS Lambda 2027

Ajout d'une rubrique Amazon SNS comme déclencheur de fonction Lambda à l'aide de la console ... 2028
Ajout manuel d'une rubrique Amazon SNS comme déclencheur de fonction Lambda ... 2028
Exemple de forme d'évènement SNS ... 2029
didacticiel ... 2030
Autorisations Lambda ... 2050
Rôle d'exécution (autorisations pour les fonctions d'accéder à d'autres ressources) ... 2052
Création d'un rôle d'exécution dans la console IAM ... 2052
Création et gestion des rôles à l'aide du AWS CLI ... 2053
Accorder un accès assorti d'un privilège minimum à votre rôle d'exécution Lambda ... 2055
Mise à jour d'un rôle d'exécution ... 2055
AWS politiques gérées ... 2057
ARN de la fonction source ... 2060
Autorisations d'accès (autorisations permettant à d'autres entités d'accéder à vos fonctions) ... 2065
Politiques basées sur l'identité ... 2065
Politiques basées sur les ressources ... 2072
Contrôle d'accès basé sur les attributs ... 2081
Ressources et conditions ... 2088
Sécurité, gouvernance et conformité ... 2096
Protection des données ... 2097
Chiffrement en transit ... 2098
Chiffrement au repos ... 2099
Utilisation des rôles liés à un service ... 2104
Autorisations de rôle liées à un service pour Lambda ... 2104
Création d'un rôle lié à un service pour Lambda ... 2105
Modification d'un rôle lié à un service pour Lambda ... 2105
Supprimer un rôle lié à un service pour Lambda ... 2106
Régions prises en charge pour les rôles liés à un service Lambda ... 2106
Gestion des identités et des accès ... 2108
Public ciblé ... 2108
Authentification par des identités ... 2109
Gestion des accès à l'aide de politiques ... 2110
Fonctionnement d'AWS Lambda avec IAM ... 2112
Exemples de politiques basées sur l'identité ... 2118
AWS politiques gérées ... 2121
Dépannage ... 2131

Gouvernance 2133
Contrôles proactifs avec Guard 2135
Contrôles proactifs avec AWS Config 2139
Contrôles de détection avec AWS Config 2147
Signature de code 2152
Analyse du code 2155
Observabilité 2160
Validation de la conformité 2169
Résilience 2169
Sécurité de l'infrastructure 2170
Sécurisation des charges de travail avec des points de terminaison publics 2171
Authentification et autorisation 2171
Protection des points de terminaison d'API 2172
Signature de code 2173
Validation de signature 2174
Créer une configuration 2175
Permissions 2177
Balises de configuration de signature de code 2178
Fonctions de surveillance et débogage 2183
Tarification 2183
Métriques de fonction 2184
Affichage des métriques de fonctions 2184
Types de métriques 2185
Journaux de fonctions 2194
Choix d'une destination de service à laquelle envoyer les journaux 2194
Configuration des destinations de journal 2195
Configuration de commandes de journalisation avancées pour votre fonction Lambda 2196
Formats des journaux 2196
Filtrage au niveau du journal 2203
Journaliser avec CloudWatch Logs 2210
Se connecter avec Firehose 2229
Connexion avec Amazon S3 2231
journaux CloudTrail 2236
Événements de données Lambda dans CloudTrail 2237
Événements de gestion Lambda dans CloudTrail 2239
Utilisation de CloudTrail pour dépanner les sources d'événement Lambda désactivées 2241

Exemples d'événements Lambda 2242
AWS X-Ray 2245
Comprendre les suivis X-Ray 2246
Comportement de suivi par défaut dans Lambda 2251
Autorisations du rôle d'exécution 2252
Activation du suivi `Active` avec l'API Lambda 2252
Activation du `Active` suivi avec CloudFormation 2252
Informations sur les fonctions 2254
Fonctionnement 2254
Tarification 2255
Environnements d'exécution pris en charge 2255
Activation de Lambda Insights dans la console 2255
Activation de Lambda Insights par programme 2256
Utilisation du tableau de bord Lambda Insights 2256
Détection d'anomalies de fonction 2258
Dépannage d'une fonction 2260
Quelle est la prochaine étape ? 2262
Affichage des métriques d'application 2263
Application Signals 2266
Mode d'intégration de la vigie applicative à Lambda 2266
Tarification 2267
Environnements d'exécution pris en charge 2267
Activation de la vigie applicative dans la console Lambda 2268
Utilisation du tableau de bord de la vigie applicative 2268
Déboguer avec VS Code 2270
Environnements d'exécution pris en charge 2270
Sécurité et débogage à distance 2270
Prérequis 2271
Débogage à distance de fonctions Lambda 2272
Désactiver le débogage à distance 2272
Informations supplémentaires 2273
Couches Lambda 2274
Utilisation des couches 2276
Couches et versions de couches 2276
Empaquetage des couches 2278
Chemins d'accès de couche pour chaque exécution Lambda 2278

Création et suppression de couches 2282
Création d'une couche 2282
Suppression d'une version de couche 2283
Ajout de couches 2285
Recherche d'informations sur la couche 2287
Couches avec CloudFormation 2289
Couches avec AWS SAM 2290
Extensions Lambda 2291
Environnement d'exécution 2292
Impact sur les performances et les ressources 2293
Autorisations 2294
Configuration des extensions 2295
Configuration des extensions (archive de fichiers .zip) 2295
Utilisation des extensions dans les images de conteneur 2295
Étapes suivantes 2296
Partenaires des extensions 2297
AWSExtensions gérées 2298
API d'extensions 2299
Cycle de vie d'un environnement d'exécution Lambda 2300
Référence d'API d'extensions 2310
API de télémétrie 2317
Création d'extensions à l'aide de l'API de télémétrie 2319
Enregistrement de votre extension 2320
Création d'un écouteur de télémétrie 2321
Spécification d'un protocole de destination 2322
Configuration de l'utilisation de la mémoire et de la mise en mémoire tampon 2323
Envoi d'une demande d'abonnement à l'API de télémétrie 2325
Messages entrants de l'API de télémétrie 2326
Référence d'API 2330
Référence du schéma `Event` 2334
Conversion d'événements en OTel Spans 2356
API Logs 2362
Résolution des problèmes 2376
Configuration 2376
Configurations de mémoire 2377
Configurations dépendant du processeur 2377

Délais 2377
Fuite de mémoire entre les invocations 2378
Résultats asynchrones renvoyés à une invocation ultérieure 2381
Déploiement 2386
Général : autorisation refusée/impossible de charger ce fichier 2387
Général : Une erreur se produit lors de l'appel du UpdateFunctionCode 2387
Amazon S3 : code d'erreur PermanentRedirect. 2388
Général : impossible de trouver, impossible de charger, impossible d’importer, classe introuvable, aucun fichier ou répertoire de ce type 2388
Général : gestionnaire de méthode non défini 2389
Général : limite de stockage du code Lambda dépassée 2389
Lambda : échec de la conversion de couche 2390
Lambda : ou InvalidParameterValueException RequestEntityTooLargeException 2391
Lambda : InvalidParameterValueException 2391
Lambda : quotas de simultanéité et de mémoire 2392
Lambda : configuration d’alias non valide pour la simultanéité provisionnée. 2392
Invocation 2393
Lambda : expiration de la fonction pendant la phase d’initialisation (Sandbox.Timedout) ... 2394
IAM : lambda : non autorisé InvokeFunction 2395
Lambda : Impossible de trouver un bootstrap valide (Runtime). InvalidEntrypoint) 2395
Lambda : L'opération ne peut pas être effectuée ResourceConflictException 2396
Lambda : La fonction est bloquée à l’état Pending (En attente) 2396
Lambda : Une fonction utilise toute la simultanéité 2396
Général : Impossible d’invoquer la fonction avec d’autres comptes ou services 2397
Général : L’invocation de la fonction boucle 2397
Lambda : Routage d’alias avec une simultanéité approvisionnée 2397
Lambda : Démarrages à froid avec la simultanéité allouée 2397
Lambda : Démarrages à froid avec de nouvelles versions 2398
Lambda : sortie inattendue de Node.js pendant l'exécution (Runtime). NodejsExit) 2399
EFS : La fonction n’a pas pu monter le système de fichiers EFS 2399
EFS : La fonction n’a pas pu se connecter au système de fichiers EFS 2400
EFS : La fonction n’a pas pu monter le système de fichiers EFS en raison d’une expiration de délai 2400
Lambda : Lambda a détecté un processus d’E/S qui prenait trop de temps 2400
Conteneur : CodeArtifactUserException erreurs 2400
Conteneur : InvalidEntrypoint erreurs 2401

Exécution 2401
Lambda : débogage à distance avec Visual Studio Code 2402
Lambda : l'exécution prend trop de temps 2403
Lambda : données utiles d'événement imprévu 2403
Lambda : données utiles plus volumineuses qu'attendu 2404
Lambda : erreurs de codage et de décodage JSON 2404
Lambda : les journaux ou les traces n'apparaissent pas 2405
Lambda : certains journaux de ma fonction n'apparaissent pas 2406
Lambda : la fonction renvoie avant la fin de l'exécution 2407
Lambda : exécution d'une version ou d'un alias de fonction imprévu 2407
Lambda : détection de boucles infinies 2408
Général : indisponibilité du service en aval 2409
AWS SDK : versions et mises à jour 2410
Python : les bibliothèques se chargent de manière incorrecte 2411
Java : votre fonction met plus de temps à traiter les événements après la mise à jour de Java 11 vers Java 17 2412
Kafka : Problèmes de gestion des erreurs et de configuration des nouvelles tentatives 2412
Mappage de source d'événement 2413
Identification et gestion de la limitation 2413
Erreurs dans la fonction de traitement 2415
Identification et gestion de la contre-pression 2417
Réseaux 2418
VPC : la fonction perd l'accès à Internet ou expire 2418
VPC : échec intermittent de la connexion TCP ou UDP 2419
VPC : la fonction doit être accessible Services AWS sans utiliser Internet 2419
VPC : limite d'interface réseau Elastic atteinte 2420
EC2: interface réseau élastique de type « lambda » 2420
DNS : échec de la connexion aux hôtes avec UNKNOWNHOSTEXCEPTION 2420
Exemples d'applications 2421
Travailler avec AWS SDKs 2424
Exemples de code 2426
Principes de base 2429
Hello Lambda 2430
Principes de base 2439
Actions 2576
Scénarios 2716

Confirmation automatique des utilisateurs connus avec une fonction Lambda 2717
Migration automatique des utilisateurs connus avec une fonction Lambda 2757
Créer une API REST pour suivre les données de la COVID-19 2781
Créer une API REST de bibliothèque de prêt 2782
Créer une application de messagerie 2783
Création d'une application sans serveur pour gérer des photos 2784
Créer une application de chat WebSocket 2788
Créez une application pour analyser les commentaires des clients 2789
Invoquer une fonction Lambda à partir d'un navigateur 2795
Transformation de données avec S3 Object Lambda 2796
Utiliser API Gateway pour invoquer une fonction Lambda 2797
Utiliser les fonctions Step Functions pour invoquer des fonctions Lambda 2799
Utilisent des événements planifiés pour invoquer une fonction Lambda 2799
Utilisation de l'API Neptune pour interroger des données de graphe 2802
Rédaction de données d'activité personnalisées à l'aide d'une fonction Lambda après authentification de l'utilisateur Amazon Cognito 2802
Exemples sans serveur 2825
Connexion à une base de données Amazon RDS dans une fonction Lambda 2826
Invoquer une fonction Lambda à partir d'un déclencheur Kinesis 2845
Invocation d'une fonction Lambda à partir d'un déclencheur DynamoDB 2856
Invocation d'une fonction Lambda à partir d'un déclencheur Amazon DocumentDB 2865
Invocation d'une fonction Lambda à partir d'un déclencheur Amazon MSK 2877
Invoquer une fonction Lambda à partir d'un déclencheur Amazon S3 2887
Invocation d'une fonction lambda à partir d'un déclencheur Amazon SNS 2899
Invoquer une fonction Lambda à partir d'un déclencheur Amazon SQS 2908
Signalement des échecs d'articles par lots pour les fonctions Lambda à l'aide d'un déclencheur Kinesis 2917
Signalement des échecs d'articles par lots pour les fonctions Lambda à l'aide d'un déclencheur DynamoDB 2930
Signalement des échecs d'articles par lots pour les fonctions Lambda à l'aide d'un déclencheur Amazon SQS 2941
AWS contributions communautaires 2951
Création et test d'une application sans serveur 2951
Quotas Lambda 2954
calcul et stockage 2955
Configuration, déploiement et exécution de fonction 2957

Requêtes d’API Lambda 2960
Autres services 2961
Historique de la documentation 2963
Mises à jour antérieures 2996
..... mmmvi

Qu'est-ce que c'est AWS Lambda ?

AWS Lambda est un service de calcul qui exécute du code sans qu'il soit nécessaire de gérer des serveurs. Votre code s'exécute automatiquement à la hausse ou à la baisse, avec une pay-per-use tarification. Consultez Création de votre première fonction pour démarrer.

Vous pouvez utiliser Lambda aux fins suivantes :

- Traitement des fichiers : traitez automatiquement les fichiers lorsqu'ils sont chargés sur Amazon Simple Storage Service. Consultez les exemples de traitement de fichiers pour plus de détails.
- Flux de travail de longue durée : utilisez des fonctions Lambda durables pour créer des flux de travail dynamiques en plusieurs étapes pouvant s'exécuter pendant un an au maximum. Parfait pour le traitement des commandes, les flux de travail d'approbation, human-in-the-loop les processus et les pipelines de données complexes qui doivent se souvenir de leur progression.
- Exemples d'opérations et d'intégration de bases de données : répondez aux modifications de base de données et automatisez les flux de données. Consultez les exemples de bases de données pour plus de détails.
- Tâches planifiées et périodiques : exécutez des opérations automatisées selon un calendrier régulier à l'aide de EventBridge. Consultez les exemples de tâches planifiées pour plus de détails.
- Traitement des flux : traitez les flux de données en temps réel à des fins d'analytique et de surveillance. Consultez Kinesis Data Streams pour plus de détails.
- Applications Web : créez des applications Web à l'échelle qui s'adaptent automatiquement à la demande.
- Backends mobiles : créez des backends d'API sécurisés pour les applications mobiles et Web.
- Backends IoT : gérez les demandes d'API Web, mobiles, IoT et tierces. Consultez IoT pour plus de détails.

Pour de plus amples informations sur la tarification, veuillez consulter AWS Lambda Pricing (français non garanti).

Fonctionnement de Lambda

Lorsque vous utilisez Lambda, vous n'êtes responsable que de votre code. Lambda exécute le code sur une infrastructure informatique à haute disponibilité et gère toutes les ressources de calcul, y

compris la maintenance des serveurs et du système d'exploitation, l'allocation et automatique de capacité, la mise à l'échelle automatique et la journalisation.

Lambda étant un service de calcul sans serveur piloté par les événements, il utilise un paradigme de programmation différent de celui des applications Web traditionnelles. Le modèle suivant illustre le fonctionnement de Lambda :

1. Vous écrivez et organisez votre code dans des fonctions Lambda, qui sont les éléments de base que vous utilisez pour créer une application Lambda.
2. Vous contrôlez la sécurité et l'accès via des autorisations Lambda, en utilisant des rôles d'exécution pour gérer les AWS services avec lesquels vos fonctions peuvent interagir et les politiques de ressources peuvent interagir avec votre code.
3. Les sources d'événements et AWS les services déclenchent vos fonctions Lambda, en transmettant les données d'événements au format JSON, que vos fonctions traitent (cela inclut les mappages de sources d'événements).
4. Lambda exécute votre code avec des environnements d'exécution spécifiques au langage (comme Node.js et Python) dans des environnements d'exécution qui empaquettent votre environnement d'exécution, vos couches et vos extensions.

> **Tip**
>
> Pour apprendre à créer des solutions sans serveur, consultez le Guide du développeur sans serveur.

Fonctions principales

Configurez, contrôlez et déployez des applications sécurisées :

- Variables d'environnement modifiez le comportement des applications sans nouveaux déploiements de code.
- Versions testez les nouvelles fonctionnalités en toute sécurité tout en maintenant des environnements de production stables.
- Couches Lambda optimisez la réutilisation et la maintenance du code en partageant des composants communs entre plusieurs fonctions.

- Signature de code appliquez la conformité en matière de sécurité en veillant à ce que seul le code approuvé atteigne les systèmes de production.

Mise à l'échelle et fonctionnement fiable :

- Les contrôles de simultanéité et de mise à l'échelle gèrent avec précision la réactivité des applications et l'utilisation des ressources lors des pics de trafic.
- Lambda SnapStart réduisez considérablement les temps de démarrage à froid. Lambda SnapStart peut fournir des performances de démarrage inférieures à une seconde, généralement sans modification de votre code de fonction.
- Streaming des réponses optimisez les performances des fonctions en fournissant progressivement des données utiles volumineuses pour un traitement en temps réel.
- Images de conteneurs empaquetez des fonctions avec des dépendances complexes à l'aide des flux de travail de conteneurs.

Connectez et intégrez en toute simplicité :

- Les réseaux VPC sécurisent les ressources sensibles et les services internes.
- Système de fichiers une intégration qui partage des données persistantes et gère les opérations dynamiques lors des invocations de fonctions.
- Fonction URLscréez des terminaux APIs et des terminaux destinés au public sans services supplémentaires.
- Extensions Lambda augmentez les fonctions grâce à des outils opérationnels, de surveillance et de sécurité.

Informations connexes

- Pour plus d'informations sur la manière dont Lambda fonctionne, consultez Fonctionnement de Lambda.
- Pour commencer à utiliser Lambda, consultez Création de votre première fonction Lambda.
- Pour obtenir une liste d'exemples d'applications, consultez Commencer avec des exemples d'applications et de modèles.

Fonctionnement de Lambda

Les fonctions Lambda sont les blocs de construction de base que vous utilisez pour créer des applications Lambda. Pour écrire des fonctions, il est essentiel de comprendre les concepts et composants fondamentaux du modèle de programmation Lambda. Cette section vous guidera à travers les éléments fondamentaux que vous devez connaître pour commencer à créer des applications sans serveur avec Lambda.

- Fonctions Lambda et gestionnaires de fonctions- Une fonction Lambda est un petit bloc de code qui s'exécute en réponse à des événements. Les fonctions peuvent être standard (jusqu'à 15 minutes) ou durables (jusqu'à un an). Les fonctions sont les éléments de base que vous utilisez pour créer des applications. Les gestionnaires de fonctions constituent le point d'entrée pour les objets d'événements traités par le code de votre fonction Lambda.
- Environnement d'exécution et exécutions Lambda : les environnements d'exécution Lambda gèrent les ressources nécessaires à l'exécution de votre fonction. Pour des fonctions durables, l'environnement d'exécution inclut des fonctionnalités de gestion automatique de l'état et de point de contrôle. Les environnements d'exécution sont les environnements spécifiques au langage dans lesquels vos fonctions s'exécutent.
- Événements et déclencheurs- D'autres Services AWS peuvent invoquer vos fonctions en réponse à des événements spécifiques. Pour des fonctions durables, les événements peuvent également déclencher la reprise des flux de travail interrompus.
- Autorisations et rôles Lambda- Contrôlez qui peut accéder à vos fonctions et avec quelles autres Services AWS fonctions peuvent interagir. Les fonctions durables nécessitent des autorisations supplémentaires pour la gestion des états et une exécution prolongée.

> **Tip**
>
> Si vous souhaitez commencer par comprendre le développement sans serveur de manière plus générale, consultez la section Comprendre la différence entre le développement traditionnel et le développement sans serveur dans le Guide du développeur AWS sans serveur.

Fonctions Lambda et gestionnaires de fonctions

Dans Lambda, les fonctions sont les blocs de construction fondamentaux que vous utilisez pour créer des applications. Une fonction Lambda est un élément de code qui s'exécute en réponse à des événements, par exemple lorsqu'un utilisateur clique sur un bouton d'un site Web ou qu'un fichier est chargé dans un compartiment Amazon Simple Storage Service (Amazon S3). Avec des fonctions durables, votre code peut suspendre l'exécution entre les étapes, en maintenant automatiquement l'état, ce qui les rend idéales pour les flux de travail de longue durée tels que le traitement des commandes ou la modération du contenu. Vous pouvez considérer une fonction comme une sorte de programme autonome doté des propriétés suivantes.

Un gestionnaire de fonction Lambda est la méthode dans le code de votre fonction qui traite les événements. Lorsqu'une fonction s'exécute en réponse à un événement, Lambda exécute le gestionnaire de fonction. Les données relatives à l'événement à l'origine de l'exécution de la fonction sont transmises directement au gestionnaire. Là où le code d'une fonction Lambda peut contenir plusieurs méthodes ou fonctions, les fonctions Lambda ne peuvent avoir qu'un seul gestionnaire.

Pour créer une fonction Lambda, vous groupez le code de votre fonction et ses dépendances dans un package de déploiement. Lambda prend en charge deux types de packages de déploiement : les archives de fichier .zip et les images conteneurs.

- Une fonction a une tâche ou un objectif spécifique
- Elle ne s'exécute que lorsque cela est nécessaire en réponse à des événements spécifiques
- Elle s'arrête automatiquement lorsqu'elle a terminé

Environnement d'exécution et exécutions Lambda

Les fonctions Lambda s'exécutent dans un environnement d'exécution sécurisé et isolé que Lambda gère pour vous. Pour des fonctions durables, l'environnement d'exécution inclut des composants supplémentaires pour la gestion des états et la coordination du flux de travail. L'environnement d'exécution gère les processus et les ressources nécessaires à l'exécution de votre fonction. Lorsqu'une fonction est invoquée pour la première fois, Lambda crée un nouvel environnement d'exécution dans lequel la fonction doit s'exécuter. Une fois l'exécution de la fonction terminée, Lambda n'arrête pas immédiatement l'environnement d'exécution ; si la fonction est à nouveau invoquée, Lambda peut réutiliser l'environnement d'exécution existant.

L'environnement d'exécution Lambda contient également un environnement d'exécution spécifique au langage (runtime), qui relaie les informations d'un événement et les réponses entre Lambda

et votre fonction. Lambda fournit un certain nombre d'environnements d'exécution gérés pour les langages de programmation les plus courants, ou vous pouvez créer les vôtres.

Pour les environnements d'exécution gérés, Lambda applique automatiquement les mises à jour de sécurité et correctifs aux fonctions lors de l'exécution.

Événements et déclencheurs

Vous pouvez également appeler une fonction Lambda directement à l'aide de la console Lambda ou de l'un des kits de développement AWS logiciel (). AWS CLISDKs Dans une application de production, il est plus courant que votre fonction soit invoquée par une autre Service AWS en réponse à un événement particulier. Par exemple, vous souhaiterez peut-être qu'une fonction s'exécute chaque fois qu'un élément est ajouté à une table Amazon DynamoDB.

Pour que votre fonction réagisse aux événements, vous devez configurer un déclencheur. Un déclencheur connecte votre fonction à une source d'événements, et votre fonction peut avoir plusieurs déclencheurs. Lorsqu'un événement se produit, Lambda reçoit les données de l'événement sous forme de document JSON et les convertit en un objet que votre code peut traiter. Vous pouvez définir le format JSON suivant pour votre événement, et l'environnement d'exécution Lambda convertit ce JSON en objet avant de le transmettre au gestionnaire de votre fonction.

Example événement lambda personnalisé

```
{
  "Location": "SEA",
  "WeatherData":{
    "TemperaturesF":{
      "MinTempF": 22,
      "MaxTempF": 78
    },
    "PressuresHPa":{
      "MinPressureHPa": 1015,
      "MaxPressureHPa": 1027
    }
  }
}
```

Pour les services de diffusion en continu et de mise en file d'attente comme Amazon Kinesis ou Amazon SQS, Lambda utilise un mappage des sources d'événements au lieu d'un déclencheur standard. Les mappages des sources d'événements interrogent la source à la recherche de nouvelles données, regroupent les enregistrements, puis invoquent votre fonction avec les

événements par lots. Pour de plus amples informations, veuillez consulter Différence entre les mappages de sources d'événements et les déclencheurs directs.

Pour comprendre le fonctionnement d'un déclencheur, commencez par suivre le didacticiel Utiliser un déclencheur Amazon S3 ou, pour obtenir une présentation générale de l'utilisation des déclencheurs et des instructions sur la création d'un déclencheur à l'aide de la console Lambda, consultez Intégration d'autres services.

Autorisations et rôles Lambda

Pour Lambda, il existe deux principaux types d'autorisations que vous devez configurer :

- Autorisations dont votre fonction a besoin pour accéder à d'autres Services AWS
- Autorisations dont les autres utilisateurs Services AWS ont besoin pour accéder à votre fonction

Les sections suivantes décrivent ces deux types d'autorisations et décrivent les meilleures pratiques pour appliquer les autorisations de moindre privilège.

Autorisations permettant aux fonctions d'accéder à d'autres AWS ressources

Les fonctions Lambda ont souvent besoin d'accéder à d'autres AWS ressources et d'effectuer des actions sur celles-ci. Par exemple, une fonction peut lire des éléments d'une table DynamoDB, stocker un objet dans un compartiment S3 ou écrire dans une file d'attente Amazon SQS. Pour donner aux fonctions les autorisations dont elles ont besoin pour effectuer ces actions, vous utilisez un rôle d'exécution.

Un rôle d'exécution Lambda est un type spécial de rôle Gestion des identités et des accès AWS (IAM), une identité que vous créez dans votre compte et à laquelle des autorisations spécifiques sont associées, définies dans une politique.

Chaque fonction Lambda doit avoir un rôle d'exécution, et un même rôle peut être utilisé par plusieurs fonctions. Lorsqu'une fonction est invoquée, Lambda assume le rôle d'exécution de la fonction et est autorisé à effectuer les actions définies dans la politique du rôle.

Lorsque vous créez une fonction dans la console Lambda, Lambda crée automatiquement un rôle d'exécution pour votre fonction. La politique du rôle donne à votre fonction les autorisations de base nécessaires pour écrire des sorties de journal dans Amazon CloudWatch Logs. Pour autoriser votre fonction à effectuer des actions sur d'autres AWS ressources, vous devez modifier le rôle afin d'ajouter les autorisations supplémentaires. Le moyen le plus simple d'ajouter des autorisations

consiste à utiliser une politique AWS gérée. Les politiques gérées sont créées et administrées par de nombreux cas d'utilisation courants AWS et fournissent des autorisations pour de nombreux cas d'utilisation courants. Par exemple, si votre fonction effectue des opérations CRUD sur une table DynamoDB, vous pouvez ajouter AmazonDynamoDBFullla politique d'accès à votre rôle.

Autorisations permettant à d'autres utilisateurs et ressources d'accéder à votre fonction

Pour accorder d'autres Service AWS autorisations d'accès à votre fonction Lambda, vous devez utiliser une politique basée sur les ressources. Dans IAM, les politiques basées sur les ressources sont attachées à une ressource (dans ce cas, votre fonction Lambda) et définissent qui peut accéder à la ressource et quelles actions peuvent être effectuées.

Pour Service AWS qu'une autre personne invoque votre fonction via un déclencheur, la politique basée sur les ressources de votre fonction doit autoriser ce service à utiliser l'`lambda:InvokeFunction`action. Si vous créez le déclencheur à l'aide de la console, Lambda ajoute automatiquement cette autorisation pour vous.

Pour autoriser d'autres AWS utilisateurs à accéder à votre fonction, vous pouvez le définir dans la politique basée sur les ressources de votre fonction exactement de la même manière que pour une autre fonction Service AWS ou ressource. Vous pouvez également utiliser une politique basée sur l'identité associée à l'utilisateur.

Bonnes pratiques relatives aux autorisations Lambda

Lorsque vous définissez des autorisations avec des stratégies IAM, la bonne pratique de sécurité est d'accorder uniquement les autorisations nécessaires à l'exécution d'une seule tâche. C'est ce que l'on appelle le principe du moindre privilège. Pour commencer à accorder des autorisations pour votre fonction, vous pouvez choisir d'utiliser une politique AWS gérée. Les politiques gérées peuvent être le moyen le plus rapide et le plus simple d'accorder des autorisations pour effectuer une tâche, mais elles peuvent également inclure d'autres autorisations dont vous n'avez pas besoin. À mesure que vous passez du stade initial du développement aux tests et à la production, nous vous recommandons de limiter les autorisations aux seules autorisations nécessaires en définissant vos propres politiques gérées par le client.

Le même principe s'applique lorsque vous accordez des autorisations d'accès à votre fonction à l'aide d'une politique basée sur les ressources. Par exemple, si vous souhaitez autoriser Amazon S3 à appeler votre fonction, la meilleure pratique consiste à limiter l'accès à des compartiments individuels, ou à des compartiments en particulier Comptes AWS, plutôt que d'accorder des autorisations générales au service S3.

Exécution de code avec Lambda

Lorsque vous écrivez une fonction Lambda, vous créez du code qui s'exécute dans un environnement sans serveur unique. Comprendre comment Lambda exécute réellement votre code implique deux aspects essentiels : le modèle de programmation, qui définit la façon dont votre code interagit avec Lambda, et le cycle de vie de l'environnement d'exécution, qui détermine la manière dont Lambda gère l'environnement d'exécution de votre code.

Le modèle de programmation Lambda

Le modèle de programmation fonctionne comme un ensemble de règles communes régissant la façon dont Lambda fonctionne avec votre code, que vous écriviez en Python, en Java ou dans tout autre langage pris en charge. Le modèle de programmation inclut votre environnement d'exécution et votre gestionnaire.

Pour les fonctions standard :

1. Lambda reçoit un événement.
2. Lambda utilise le runtime pour préparer l'événement dans un format que votre code peut utiliser.
3. L'environnement d'exécution envoie l'événement formaté à votre gestionnaire.
4. Votre gestionnaire traite l'événement en utilisant le code que vous avez écrit.

Pour des fonctions durables :

1. Lambda reçoit un événement
2. Le runtime prépare à la fois l'événement et DurableContext
3. Votre gestionnaire peut :
   - Étapes du processus avec point de contrôle automatique
   - Interrompre l'exécution sans consommer de ressources
   - CV du dernier point de contrôle réussi
   - Maintenir l'état entre les étapes

Le gestionnaire, auquel Lambda envoie des événements à traiter par votre code, est essentiel à ce modèle. Considérez-le comme le point d'entrée de votre code. Lorsque Lambda reçoit un événement, il le transmet à votre gestionnaire avec certaines informations contextuelles. Le gestionnaire exécute

ensuite votre code pour traiter ces événements. Par exemple, il peut lire un fichier lorsqu'il est chargé sur Amazon S3, analyser une image ou mettre à jour une base de données. Une fois que votre code a fini de traiter un événement, le gestionnaire est prêt à traiter le suivant.

Le modèle d'exécution Lambda

Alors que le modèle de programmation définit la manière dont Lambda interagit avec votre code, l'environnement d'exécution est l'endroit où Lambda exécute réellement votre fonction. Il s'agit d'un espace de calcul sécurisé et isolé créé spécifiquement pour votre fonction.

Chaque environnement suit un cycle de vie qui varie entre des fonctions standard et durables :

Fonctions standard (jusqu'à 15 minutes) :

1. Initialisation : configuration de l'environnement et chargement du code
2. Invocation : exécution unique du code de fonction
3. Arrêt : nettoyage de l'environnement

Fonctions durables (jusqu'à 1 an) :

1. Initialisation : configuration de l'environnement et de l'état durable
2. Invocation : plusieurs étapes avec point de contrôle automatique
3. États d'attente : pause de l'exécution sans consommation de ressources
4. Reprise : Redémarrer depuis le dernier point de contrôle
5. Arrêt : nettoyage de l'état durable

Cet environnement gère les aspects importants de l'exécution de votre fonction. Il fournit à votre fonction de la mémoire et un répertoire `/tmp` pour le stockage temporaire. Pour les fonctions durables, il gère également :

- Persistance automatique de l'état entre les étapes
- Stockage et restauration aux points de contrôle
- Coordination de l'état d'attente
- Suivi des progrès réalisés dans le cadre des exécutions de longue durée

Présentation du modèle de programmation Lambda

Lambda propose deux modèles de programmation : les fonctions standard qui s'exécutent jusqu'à 15 minutes et les fonctions durables qui peuvent fonctionner jusqu'à un an. Bien que les deux partagent des concepts fondamentaux, les fonctions durables ajoutent des fonctionnalités pour des flux de travail dynamiques et de longue durée.

Lambda fournit un modèle de programmation commun à tous les runtimes. Le modèle de programmation définit l'interface entre votre code et le système Lambda. Vous indiquez à Lambda le point d'entrée de votre fonction en définissant un gestionnaire dans la configuration de la fonction. Le runtime transmet les objets au gestionnaire qui contiennent l'événement d'invocation et le contexte, tels que le nom de la fonction et l'ID de la demande.

Pour Durable Functions, le gestionnaire reçoit également un DurableContext objet qui fournit :

- Capacités de point de contrôle via step ()
- Gestion de l'état d'attente via wait () et waitForCallback ()
- Persistance automatique de l'état entre les invocations

Lorsque le gestionnaire termine le traitement du premier événement, le runtime lui en envoie un autre. Pour les fonctions durables, le gestionnaire peut suspendre l'exécution entre les étapes, et Lambda enregistrera et rétablira automatiquement l'état lorsque la fonction reprendra. La classe de la fonction reste en mémoire, de sorte que les clients et variables déclarés en dehors de la méthode du gestionnaire dans le code d'initialisation peuvent être réutilisés. Pour économiser du temps de traitement sur les événements suivants, créez des ressources réutilisables telles que des clients AWS SDK lors de l'initialisation. Une fois initialisée, chaque instance de votre fonction peut traiter des milliers de demandes.

Votre fonction a également accès au stockage local dans le répertoire `/tmp`, un cache temporaire qui peut servir à plusieurs appels. Pour de plus amples informations, veuillez consulter Environnement d’exécution.

Lorsque le suivi AWS X-Ray est activé, le runtime enregistre des sous-segments distincts pour l'initialisation et l'exécution.

Le moteur d'exécution capture les résultats de journalisation de votre fonction et les envoie à Amazon CloudWatch Logs. En plus de consigner la sortie de votre fonction, le runtime consigne également les entrées lorsque la fonction d’invocation démarre et se termine. Cela inclut un journal de rapport avec

l'ID de demande, la durée facturée, la durée d'initialisation et d'autres détails. Si votre fonction génère une erreur, le runtime renvoie cette erreur au mécanisme d'appel.

> **Note**
>
> La journalisation est soumise à des quotas de CloudWatch journalisation. Les données des journaux peuvent être perdues en raison de la limitation ou, dans certains cas, lorsqu'une instance de votre fonction est arrêtée.

Principales différences en matière de fonctions durables :

- L'état est automatiquement conservé entre les étapes
- Les fonctions peuvent suspendre l'exécution sans consommer de ressources
- Les étapes sont automatiquement réessayées en cas d'échec
- Les progrès sont suivis par le biais de points de contrôle

Lambda met à l'échelle votre fonction en exécutant des instances supplémentaires à mesure que la demande augmente et en arrêtant des instances à mesure que la demande diminue. Ce modèle entraîne des variations dans l'architecture des applications, telles que :

- Sauf indication contraire, les demandes entrantes peuvent être traitées dans le désordre ou simultanément.
- Ne comptez pas sur la longévité des instances de votre fonction, stockez plutôt l'état de votre application ailleurs.
- Utilisez le stockage local et les objets de niveau classe afin d'améliorer les performances, mais réduisez au minimum la taille de votre package de déploiement et la quantité de données que vous transférez vers l'environnement d'exécution.

Pour une introduction pratique au modèle de programmation dans le langage de programmation de votre choix, consultez les sections suivantes.

- Création de fonctions Lambda avec Node.js
- Création de fonctions Lambda avec Python
- Création de fonctions Lambda avec Ruby
- Création de fonctions Lambda avec Java

- Création de fonctions Lambda avec Go
- Création de fonctions Lambda avec C#
- Création de fonctions Lambda avec PowerShell

Comprendre le cycle de vie de l'environnement d'exécution Lambda

Les environnements d'exécution Lambda prennent en charge à la fois les fonctions standard (jusqu'à 15 minutes) et les fonctions durables (jusqu'à un an). Bien que les deux aient le même cycle de vie de base, les fonctions durables ajoutent des fonctionnalités de gestion des états pour les flux de travail de longue durée.

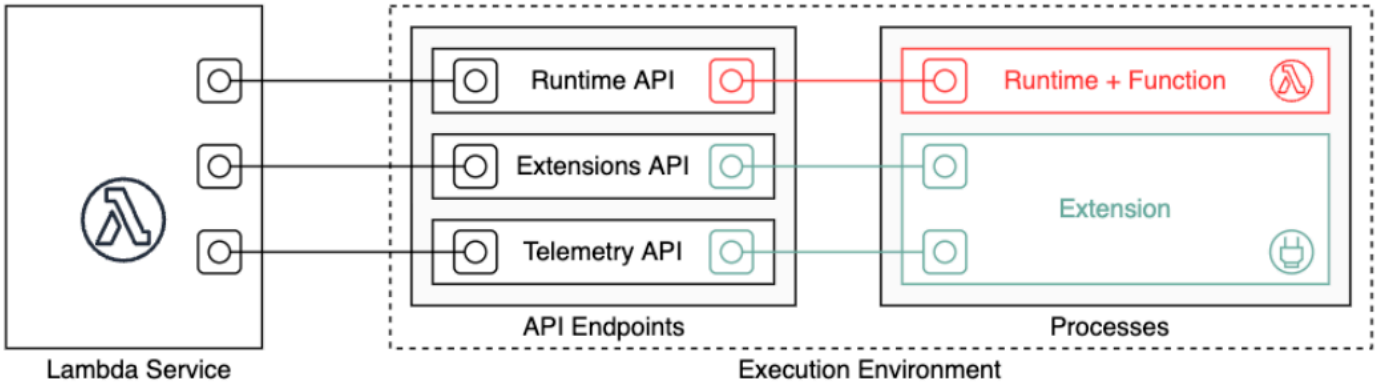
Lambda invoque votre fonction dans un environnement d'exécution, qui fournit un environnement d'exécution sécurisé et isolé. L'environnement d'exécution gère les ressources nécessaires à l'exécution de votre fonction. L'environnement d'exécution prend également en charge le cycle de vie pour l'exécution de la fonction et pour toutes les extensions externes associées à votre fonction.

Pour Durable Functions, l'environnement d'exécution inclut des composants supplémentaires pour :

- Persistance de l'état entre les étapes
- Gestion des points de contrôle
- Coordination de l'état d'attente
- Suivi des progrès

> **Environnement d'exécution des instances gérées Lambda**
>
> Si vous utilisez des instances gérées Lambda, l'environnement d'exécution présente des différences importantes par rapport aux fonctions Lambda (par défaut). Les instances gérées prennent en charge les appels simultanés, utilisent un modèle de cycle de vie différent et s'exécutent sur une infrastructure appartenant au client. Pour des informations détaillées sur l'environnement d'exécution des instances gérées, consultezComprendre l'environnement d'exécution des instances gérées Lambda.

L'exécution de la fonction communique avec Lambda à l'aide de l'API d'exécution. Les extensions communiquent avec Lambda à l'aide de l'API d'extensions. Les extensions peuvent également recevoir des messages de journal et d'autres télémétries de la fonction en utilisant l'API de télémétrie.



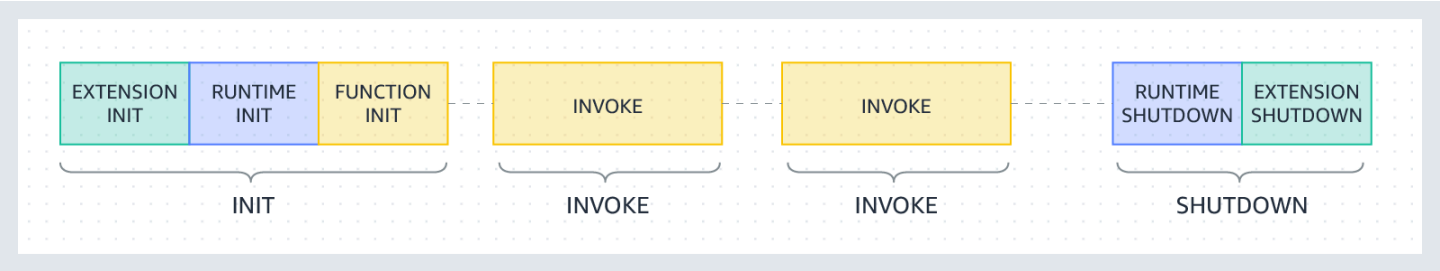
Lorsque vous créez votre fonction Lambda, vous précisez des informations de configuration, telles que la quantité de mémoire disponible et le temps d'exécution maximum autorisé pour votre fonction. Lambda utilise ces informations pour configurer l'environnement d'exécution.

L'exécution de la fonction et chaque extension externe sont des processus qui s'exécutent dans l'environnement d'exécution. Les autorisations, les ressources, les informations d'identification et les variables d'environnement sont partagées entre la fonction et les extensions.

Rubriques

- Cycle de vie d'un environnement d'exécution Lambda
- Démarrages à froid et latence
- Réduction des démarrages à froid avec la simultanéité provisionnée
- Optimisation de l'initialisation statique

Cycle de vie d'un environnement d'exécution Lambda



Chaque phase commence par un événement que Lambda envoie à l'exécution et à toutes les extensions enregistrées. L'exécution et chaque extension indiquent la fin de l'opération en envoyant une demande d'API `Next`. Lambda gèle l'environnement d'exécution lorsque l'exécution de chaque extension est terminée, et qu'il n'y a pas d'événement en attente.

Les phases du cycle de vie de Durable Functions incluent :

- Init : initialisation standard et configuration d'état durable
- Invoke : peut inclure des exécutions en plusieurs étapes avec pointage automatique
- Attendre : la fonction peut suspendre l'exécution sans consommer de ressources
- Reprise : la fonction redémarre depuis le dernier point de contrôle
- Arrêt : nettoyage de l'état durable et des ressources

Rubriques

- Phase d’initialisation
- Échecs pendant la phase d’initialisation
- Phase de restauration (Lambda uniquement SnapStart)
- Phase d’invocation
- Échecs pendant la phase d’invocation
- Phase d’arrêt

Phase d’initialisation

Dans la phase `Init`, Lambda effectue trois tâches :

- Démarrage de toutes les extensions (`Extension init`)
- Amorçage de l’exécution (`Runtime init`)
- Exécution du code statique de la fonction (`Function init`)
- Exécutez tous les hooks d'exécution avant le point de contrôle (Lambda uniquement) SnapStart

La phase `Init` se termine lorsque l’exécution et toutes les extensions signalent qu’elles sont prêtes en envoyant une demande d’API `Next`. La phase `Init` est limitée à 10 secondes. Si les trois tâches ne se terminent pas dans les 10 secondes, Lambda relance la phase `Init` au moment de la première invocation de fonction avec le délai d’attente de fonction configuré.

Lorsque Lambda SnapStart est activé, la phase `Init` se produit lorsque vous publiez une version de la fonction. Lambda enregistre un instantané de l’état de la mémoire et du disque de l’environnement d’exécution initialisé, fait persister l’instantané chiffré et le met en cache pour un accès à faible

latence. Si vous avez un hook d'environnement d'exécution avant le point de contrôle, le code s'exécute alors à la fin de la phase `Init`.

> **Note**
>
> Le délai d'expiration de 10 secondes ne s'applique pas aux fonctions qui utilisent la simultanéité provisionnée SnapStart ou les instances gérées Lambda. Pour les fonctions de simultanéité provisionnée et d'instances gérées, votre code d'initialisation peut s'exécuter pendant 15 minutes maximum. SnapStart Le délai d'attente est de 130 secondes ou le délai d'expiration de la fonction configurée (900 secondes au maximum), la valeur la plus élevée étant retenue.

Lorsque vous utilisez la simultanéité provisionnée, Lambda initialise l'environnement d'exécution lorsque vous configurez les paramètres du PC pour une fonction. Lambda garantit également que les environnements d'exécution initialisés sont toujours disponibles avant les invocations. Vous pouvez constater des écarts entre les phases d'initialisation et d'invocation de votre fonction. En fonction de la configuration d'environnement d'exécution et de mémoire de votre fonction, vous pouvez également constater une variabilité de latence est possible lors de la première invocation dans un environnement d'exécution initialisé.

Pour les fonctions utilisant la simultanéité à la demande, Lambda peut occasionnellement initialiser les environnements d'exécution avant les demandes d'invocation. Lorsque cela se produit, vous pouvez également observer un intervalle de temps inattendu entre les phases d'initialisation et d'invocation de votre fonction. Nous vous recommandons de ne pas dépendre de ce comportement.

Échecs pendant la phase d'initialisation

Si une fonction se bloque ou expire pendant la phase `Init`, Lambda émet des informations d'erreur dans le journal `INIT_REPORT`.

Example — Journal INIT_REPORT pour le délai d'expiration

```
INIT_REPORT Init Duration: 1236.04 ms Phase: init Status: timeout
```

Example — Journal INIT_REPORT en cas d'échec de l'extension

```
INIT_REPORT Init Duration: 1236.04 ms Phase: init Status: error Error Type:
 Extension.Crash
```

Si la `Init` phase est réussie, Lambda n'émet le `INIT_REPORT` journal que si la simultanéité SnapStartprovisionnée est activée. SnapStart et les fonctions de simultanéité provisionnées émettent toujours. `INIT_REPORT` Pour de plus amples informations, veuillez consulter Surveillance pour Lambda SnapStart.

Phase de restauration (Lambda uniquement SnapStart)

Lorsque vous appelez une SnapStartfonction pour la première fois et que celle-ci évolue, Lambda reprend les nouveaux environnements d'exécution à partir de l'instantané persistant au lieu d'initialiser la fonction à partir de zéro. Si vous avez un hook d'exécution après restauration, le code s'exécute à la fin de la phase `Restore`. Vous êtes facturé pour la durée des hooks d'exécution après restauration. L'exécution doit se charger et les hooks d'exécution après restauration doivent se terminer dans le délai imparti (10 secondes). Sinon, vous obtiendrez un SnapStartTimeoutException. Lorsque la phase `Restore` se termine, Lambda invoque le gestionnaire de fonction (Phase d'invocation).

Échecs pendant la phase de restauration

Si la phase `Restore` échoue, Lambda émet des informations d'erreur dans le journal `RESTORE_REPORT`.

Example — Journal RESTORE_REPORT pour le délai d'expiration

```
RESTORE_REPORT Restore Duration: 1236.04 ms Status: timeout
```

Example — Journal RESTORE_REPORT en cas d'échec du hook d'exécution

```
RESTORE_REPORT Restore Duration: 1236.04 ms Status: error Error Type: Runtime.ExitError
```

Pour plus d'informations sur le journal `RESTORE_REPORT`, consultez Surveillance pour Lambda SnapStart.

Phase d'invocation

Quand une fonction Lambda est invoquée en réponse à une demande d'API `Next`, Lambda envoie un événement `Invoke` à l'exécution et à chaque extension.

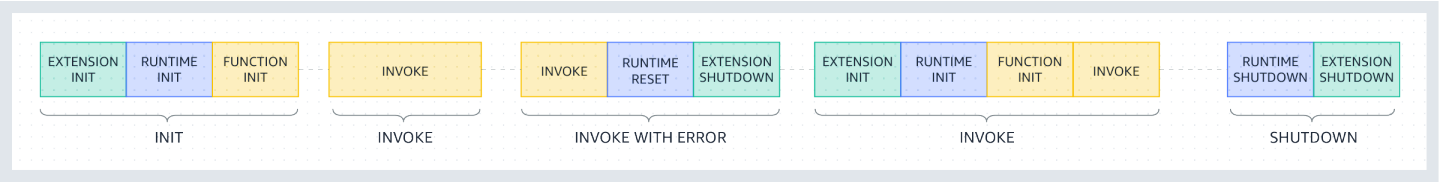
Le paramètre d'expiration de la fonction limite la durée de l'ensemble de la phase `Invoke`. Par exemple, si vous définissez le délai d'expiration de la fonction sur 360 secondes, la fonction et toutes les extensions doivent être terminées dans un délai de 360 secondes. Notez qu'il n'y a pas de phase post-invocation indépendante. La durée correspond à la somme de tous les temps d'invocation

(exécution + extensions) et n'est calculée que lorsque l'exécution de la fonction et de toutes les extensions est terminée.

La phase d'invocation prend fin lorsque l'exécution et toutes les extensions signalent qu'ils ont terminé en envoyant une demande d'API `Next`.

Échecs pendant la phase d'invocation

Si la fonction Lambda se bloque ou expire pendant la phase `Invoke`, Lambda réinitialise l'environnement d'exécution. Le diagramme suivant illustre le comportement de l'environnement d'exécution Lambda en cas d'échec de l'invocation :



Dans le diagramme précédent :

- La première phase est la phase INIT, qui s'exécute sans erreur.
- La deuxième phase est la phase INVOKE, qui s'exécute sans erreur.
- Supposons qu'à un moment donné, votre fonction rencontre un échec d'invocation (les causes courantes incluent les expirations de délai des fonctions, les erreurs d'exécution, l'épuisement de la mémoire, les problèmes de connectivité VPC, les erreurs d'autorisation, les limites de simultanéité et divers problèmes de configuration). Pour connaître la liste complète des échecs d'invocation possibles, consultez the section called "Invocation". La troisième phase, intitulée INVOKE WITH ERROR, illustre ce scénario. Lorsque cela se produit, le service Lambda effectue une réinitialisation. La réinitialisation se comporte comme un événement `Shutdown`. Lambda commence par arrêter l'exécution, puis envoie un événement `Shutdown` à chaque extension externe enregistrée. L'événement comprend le motif de l'arrêt. Si cet environnement est utilisé pour une nouvelle invocation, Lambda réinitialise l'extension et l'exécution en même temps que l'invocation suivante.

  Veuillez noter que la réinitialisation Lambda n'efface pas le contenu du répertoire `/tmp` avant la phase d'initialisation suivante. Ce comportement est cohérent avec la phase d'arrêt normale.

  > **Note**
  >
  > AWS met actuellement en œuvre des modifications du service Lambda. En raison de ces modifications, vous pouvez constater des différences mineures entre la structure

et le contenu des messages du journal système et des segments de suivi émis par les différentes fonctions Lambda de votre Compte AWS.
Si la configuration du journal système de votre fonction est définie sur du texte brut, cette modification affecte les messages de journal capturés dans CloudWatch les journaux lorsque votre fonction rencontre un échec d'appel. Les exemples suivants montrent les sorties des journaux dans les anciens et les nouveaux formats.
Ces modifications seront mises en œuvre au cours des prochaines semaines, et toutes les fonctions, Régions AWS sauf en Chine et dans les GovCloud régions, seront transférées pour utiliser le nouveau format des messages de journal et des segments de trace.

Example CloudWatch Journalise la sortie du journal (exécution ou crash de l'extension) - ancien style

```
START RequestId: c3252230-c73d-49f6-8844-968c01d1e2e1 Version: $LATEST
RequestId: c3252230-c73d-49f6-8844-968c01d1e2e1 Error: Runtime exited without
 providing a reason
Runtime.ExitError
END RequestId: c3252230-c73d-49f6-8844-968c01d1e2e1
REPORT RequestId: c3252230-c73d-49f6-8844-968c01d1e2e1 Duration: 933.59 ms Billed
 Duration: 934 ms Memory Size: 128 MB Max Memory Used: 9 MB
```

Example CloudWatch Enregistre la sortie du journal (délai d'expiration de la fonction) - ancien style

```
START RequestId: b70435cc-261c-4438-b9b6-efe4c8f04b21 Version: $LATEST
2024-03-04T17:22:38.033Z b70435cc-261c-4438-b9b6-efe4c8f04b21 Task timed out after
 3.00 seconds
END RequestId: b70435cc-261c-4438-b9b6-efe4c8f04b21
REPORT RequestId: b70435cc-261c-4438-b9b6-efe4c8f04b21 Duration: 3004.92 ms Billed
 Duration: 3117 ms Memory Size: 128 MB Max Memory Used: 33 MB Init Duration: 111.23
 ms
```

Le nouveau format des CloudWatch journaux inclut un `status` champ supplémentaire dans la `REPORT` ligne. Dans le cas d'un incident d'exécution ou d'extension, la ligne `REPORT` inclut également un champ `ErrorType`.

Example CloudWatch Journalise la sortie du journal (exécution ou crash de l'extension) - nouveau style

```
START RequestId: 5b866fb1-7154-4af6-8078-6ef6ca4c2ddd Version: $LATEST
END RequestId: 5b866fb1-7154-4af6-8078-6ef6ca4c2ddd
REPORT RequestId: 5b866fb1-7154-4af6-8078-6ef6ca4c2ddd Duration: 133.61 ms Billed
 Duration: 214 ms Memory Size: 128 MB Max Memory Used: 31 MB Init Duration: 80.00
 ms Status: error Error Type: Runtime.ExitError
```

Example CloudWatch Enregistre la sortie du journal (délai d'expiration de la fonction) - nouveau style

```
START RequestId: 527cb862-4f5e-49a9-9ae4-a7edc90f0fda Version: $LATEST
END RequestId: 527cb862-4f5e-49a9-9ae4-a7edc90f0fda
REPORT RequestId: 527cb862-4f5e-49a9-9ae4-a7edc90f0fda Duration: 3016.78 ms Billed
 Duration: 3101 ms Memory Size: 128 MB Max Memory Used: 31 MB Init Duration: 84.00
 ms Status: timeout
```

- La quatrième phase représente la phase INVOKE qui suit immédiatement un échec de l'invocation. Ici, Lambda initialise à nouveau l'environnement en relançant la phase INIT. Cela s'appelle une init supprimée. Lorsque des initialisations supprimées se produisent, Lambda ne signale pas explicitement une phase d'initialisation supplémentaire dans Logs. CloudWatch Au lieu de cela, vous pouvez remarquer que la durée dans la ligne REPORT inclut une durée INIT supplémentaire + la durée INVOKE. Supposons, par exemple, que les connexions suivantes s'affichent CloudWatch :

```
2022-12-20T01:00:00.000-08:00 START RequestId: XXX Version: $LATEST
2022-12-20T01:00:02.500-08:00 END RequestId: XXX
2022-12-20T01:00:02.500-08:00 REPORT RequestId: XXX Duration: 3022.91 ms
Billed Duration: 3000 ms Memory Size: 512 MB Max Memory Used: 157 MB
```

Dans cet exemple, la différence entre les horodatages REPORT et START est de 2,5 secondes. Cela ne correspond pas à la durée rapportée de 3022,91 millisecondes, car cela ne prend pas en compte l'INIT supplémentaire (init supprimé) que Lambda a effectué. Dans cet exemple, vous pouvez en déduire que la phase INVOKE réelle a duré 2,5 secondes.

Pour plus d'informations sur ce comportement, vous pouvez utiliser la Accès aux données de télémétrie en temps réel pour les extensions à l'aide de l'API de télémétrie. L'API de

télémétrie émet des événements `INIT_START`, `INIT_RUNTIME_DONE` et `INIT_REPORT` avec `phase=invoke` chaque fois que des inits supprimés se produisent entre les phases d'invocation.

- La cinquième phase représente la phase SHUTDOWN, qui s'exécute sans erreur.

Phase d'arrêt

Quand Lambda est sur le point d'arrêter l'exécution, il envoie un événement `Shutdown` à chaque extension externe enregistrée. Les extensions peuvent utiliser ce temps pour les tâches de nettoyage final. L'événement `Shutdown` est une réponse à une `Next`demande d'API.

Limite de durée : la durée maximale de la phase `Shutdown` dépend de la configuration des extensions enregistrées :

- 0 ms : fonction sans extension enregistrée
- 500 ms : fonction avec une extension interne enregistrée.
- 2 000 ms : fonction avec une ou plusieurs extensions externes enregistrées.

Si l'environnement d'exécution ou une extension ne répondent pas à l'événement `Shutdown` dans cette limite de temps, Lambda met fin au processus à l'aide d'un signal `SIGKILL`.

Lorsque la fonction et toutes les extensions ont pris fin, Lambda conserve l'environnement d'exécution pendant un certain temps en prévision d'une autre invocation de fonction. Cependant, Lambda met fin aux environnements d'exécution toutes les quelques heures pour permettre les mises à jour et la maintenance de l'environnement d'exécution, même pour les fonctions invoquées en continu. Vous ne devez pas partir du principe que l'environnement d'exécution sera conservé indéfiniment. Pour de plus amples informations, veuillez consulter Implémenter l'absence d'état dans les fonctions.

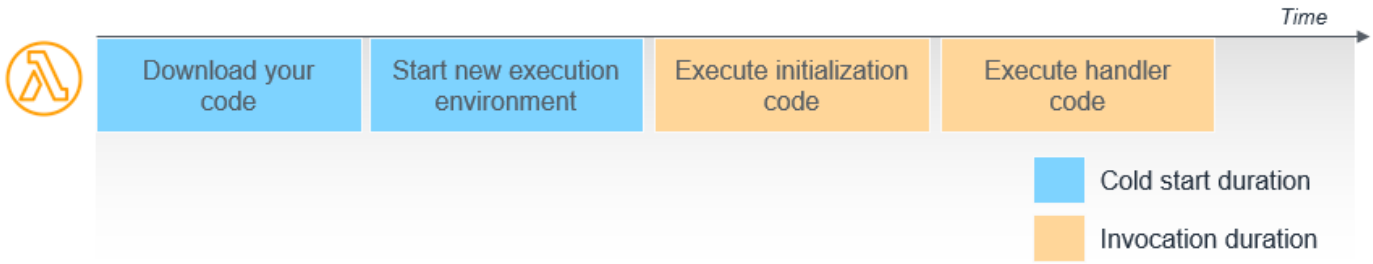
Lorsque la fonction est à nouveau invoquée, Lambda décongèle l'environnement pour le réutiliser. La réutilisation de l'environnement d'exécution a les conséquences suivantes :

- Les objets déclarés en dehors de la méthode de gestionnaire de la fonction restent initialisés, ce qui fournit une optimisation supplémentaire lorsque la fonction est invoquée à nouveau. Par exemple, si votre fonction Lambda établit une connexion de base de données, au lieu de rétablir la connexion, la connexion d'origine est utilisée dans les invocations suivantes. Nous vous recommandons d'ajouter une logique dans votre code pour vérifier s'il existe une connexion avant d'en créer une nouvelle.

- Chaque environnement d'exécution fournit entre 512 Mo et 10 240 Mo, par incréments de 1 Mo d'espace disque dans le répertoire `/tmp`. Le contenu du répertoire est conservé lorsque l'environnement d'exécution est gelé, fournissant ainsi un cache temporaire qui peut servir à plusieurs invocations. Vous pouvez ajouter du code pour vérifier si le cache contient les données que vous avez stockées. Pour de plus amples informations sur les limites de taille de déploiement, veuillez consulter Quotas Lambda.
- Les processus en arrière-plan ou les rappels qui ont été initiés par votre fonction Lambda et qui ne sont pas terminés à la fin de l'exécution de la fonction reprennent si Lambda réutilise l'environnement d'exécution. Assurez-vous que les processus d'arrière-plan ou les rappels dans votre code se terminent avant que l'exécution du code ne prenne fin.

Démarrages à froid et latence

Lorsque Lambda reçoit une demande d'exécution d'une fonction via l'API Lambda, il prépare d'abord un environnement d'exécution. Au cours de cette phase d'initialisation, le service télécharge votre code, démarre l'environnement et exécute tout code d'initialisation en dehors du gestionnaire principal. Enfin, Lambda exécute le code du gestionnaire.



Dans ce schéma, les deux premières étapes de téléchargement du code et de configuration de l'environnement sont souvent appelées « démarrage à froid ». Cette durée vous est facturée, et ajoute de la latence à la durée globale de votre invocation.

Une fois l'invocation terminée, l'environnement d'exécution est gelé. Pour améliorer la gestion des ressources et les performances, Lambda retient l'environnement d'exécution pendant une période. Pendant ce temps, si une autre demande arrive pour la même fonction, Lambda peut réutiliser l'environnement. Cette deuxième demande se termine généralement plus rapidement, étant donné que l'environnement d'exécution est déjà entièrement configuré. C'est ce qu'on appelle un « démarrage à chaud ».

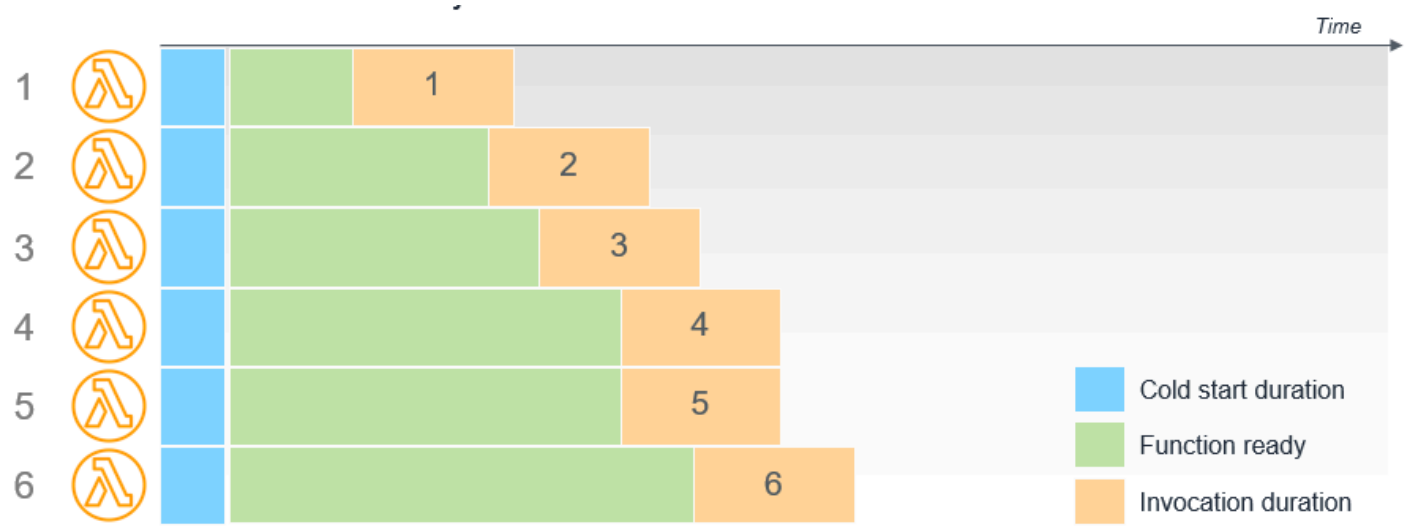
Les démarrages à froid concernent moins de 1 % des invocations. La durée d'un démarrage à froid varie de moins de 100 ms à plus d'une seconde. Les démarrages à froid concernent plus souvent les

fonctions de développement et de test que les charges de travail de production. Cela est dû au fait que les fonctions de développement et de test sont généralement invoquées moins fréquemment.

Réduction des démarrages à froid avec la simultanéité provisionnée

Si vous avez besoin d'heures de démarrage des fonctions prévisibles pour votre charge de travail, la simultanéité provisionnée est la solution recommandée pour garantir la latence la plus faible possible. Cette fonctionnalité pré-initialise les environnements d'exécution, réduisant ainsi les démarrages à froid.

Par exemple, une fonction avec une simultanéité provisionnée de 6 dispose de 6 environnements d'exécution préparés.



Optimisation de l'initialisation statique

L'initialisation statique se produit avant que le code du gestionnaire ne commence à s'exécuter dans une fonction. Il s'agit du code d'initialisation que vous fournissez ; il se trouve en dehors du gestionnaire principal. Ce code est souvent utilisé pour importer des bibliothèques et des dépendances, définir les configurations et initialiser les connexions à d'autres services.

L'exemple Python suivant montre l'importation et la configuration de modules, ainsi que la création du client Amazon S3 pendant la phase d'initialisation, avant que la fonction `lambda_handler` ne s'exécute pendant l'invocation.

```
import os
import json
import cv2
```

```
import logging
import boto3

s3 = boto3.client('s3')
logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):

  # Handler logic...
```

Le principal facteur de latence avant l'exécution d'une fonction provient du code d'initialisation. Ce code s'exécute lorsqu'un nouvel environnement d'exécution est créé pour la première fois. Le code d'initialisation n'est pas exécuté à nouveau si une invocation utilise un environnement d'exécution chaud. Les facteurs qui influent sur la latence du code d'initialisation sont notamment les suivants :

- la taille du package de fonctions, en termes de bibliothèques et de dépendances importées, et de couches Lambda ;
- la quantité de code et la tâche d'initialisation ;
- la performance des bibliothèques et des autres services lors de la configuration de connexions et d'autres ressources.

Les développeurs peuvent prendre un certain nombre de mesures pour optimiser la latence d'initialisation statique. Si une fonction comporte de nombreux objets et connexions, vous pouvez peut-être réarchitecturer une seule fonction en plusieurs fonctions spécialisées, chacune plus petite et contenant chacun moins de code d'initialisation.

Il est important que les fonctions n'importent que les bibliothèques et les dépendances dont elles ont besoin. Par exemple, si vous utilisez Amazon DynamoDB uniquement dans AWS le SDK, vous pouvez avoir besoin d'un service individuel au lieu de l'intégralité du SDK. Voici trois exemples de comparaison :

```
// Instead of const AWS = require('aws-sdk'), use:
const DynamoDB = require('aws-sdk/clients/dynamodb')

// Instead of const AWSXRay = require('aws-xray-sdk'), use:
const AWSXRay = require('aws-xray-sdk-core')

// Instead of const AWS = AWSXRay.captureAWS(require('aws-sdk')), use:
const dynamodb = new DynamoDB.DocumentClient()
```

```
AWSXRay.captureAWSClient(dynamodb.service)
```

L’initialisation statique est aussi souvent le meilleur moment pour ouvrir des connexions à une base de données afin de permettre à une fonction de réutiliser les connexions via plusieurs invocations dans le même environnement d’exécution. Cependant, il se peut que vous disposiez d’un grand nombre d’objets qui ne sont utilisés que dans certains chemins d’exécution de votre fonction. Dans ce cas, vous pouvez charger de manière différée des variables dans la portée globale afin de réduire la durée d’initialisation statique.

Évitez les variables globales pour les informations spécifiques au contexte. Si votre fonction possède une variable globale qui n’est utilisée que pendant la durée de vie d’une seule invocation et qui est réinitialisée pour la prochaine invocation, utilisez une portée de variable locale au gestionnaire. Cela permet non seulement d’éviter les fuites de variables globales lors des invocations, mais également d’améliorer les performances d’initialisation statique.

Création d’architectures guidées par les événements avec Lambda

Un événement est tout ce qui déclenche l’exécution d’une fonction Lambda. Les événements peuvent déclencher une fonction Lambda de deux manières : par invocation directe (push) et par mappage des sources d'événements (pull).

De nombreux AWS services peuvent appeler directement vos fonctions Lambda. Ces services transmettent (push) des événements à votre fonction Lambda. Les événements qui déclenchent des fonctions peuvent être pratiquement n'importe quoi, qu'il s'agisse d'une requête HTTP via API Gateway, d'un calendrier géré par une EventBridge règle, d'un AWS IoT événement ou d'un événement Amazon S3. Grâce au mappage des sources d’événements, Lambda récupère (pull) activement les événements d’une file d’attente ou d’un flux. Vous configurez Lambda pour vérifier les événements provenant d’un service pris en charge, et Lambda gère l’interrogation et l’invocation de votre fonction.

Lorsqu’ils sont transmis à votre fonction, les événements sont structurés au format JSON. La structure JSON varie en fonction du service qui la génère et du type d’événement. Alors que les appels de fonctions Lambda standard peuvent durer jusqu'à 15 minutes (ou jusqu'à un an avec des fonctions durables), Lambda convient parfaitement aux appels courts d'une seconde ou moins. Cela est particulièrement vrai pour les architectures axées sur les événements, où chaque fonction Lambda est traitée comme un microservice chargé d’exécuter un ensemble restreint d’instructions spécifiques.

> **Note**
>
> Les architectures guidées par les événements communiquent entre différents systèmes à l’aide de réseaux, ce qui introduit une latence variable. Pour les charges de travail nécessitant une très faible latence, telles que les systèmes de trading en temps réel, cette conception n'est peut-être pas le meilleur choix. Toutefois, pour les charges de travail hautement évolutives et disponibles, ou celles dont les modèles de trafic sont imprévisibles, les architectures guidées par les événements peuvent constituer un moyen efficace de répondre à ces demandes.

Rubriques

- Avantages des architectures guidées par les événements
- Compromis des architectures guidées par les événements

- Anti-modèles dans les applications pilotées par les événements basées sur Lambda

Avantages des architectures guidées par les événements

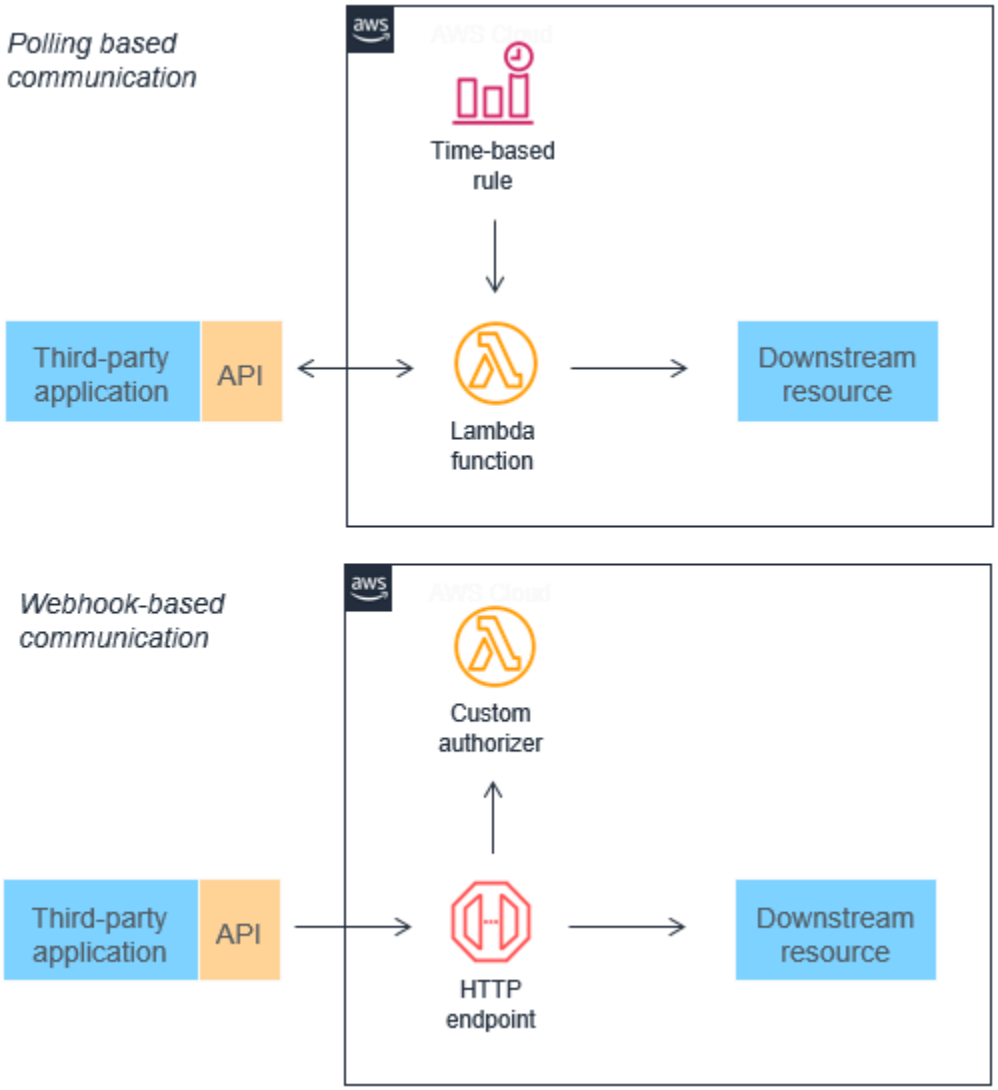
Lambda prend en charge deux méthodes d'invocation dans les architectures pilotées par les événements :

1. Invocation directe (méthode push) : les AWS services déclenchent directement les fonctions Lambda. Par exemple :
   - Amazon S3 déclenche une fonction lorsqu'un fichier est chargé
   - API Gateway déclenche une fonction lors de la réception d'une requête HTTP
2. Mappage des sources d'événements (méthode pull) : Lambda récupère les événements et invoque des fonctions. Par exemple :
   - Lambda récupère les messages d'une file d'attente Amazon SQS et invoque une fonction
   - Lambda lit les enregistrements d'un flux DynamoDB et invoque une fonction

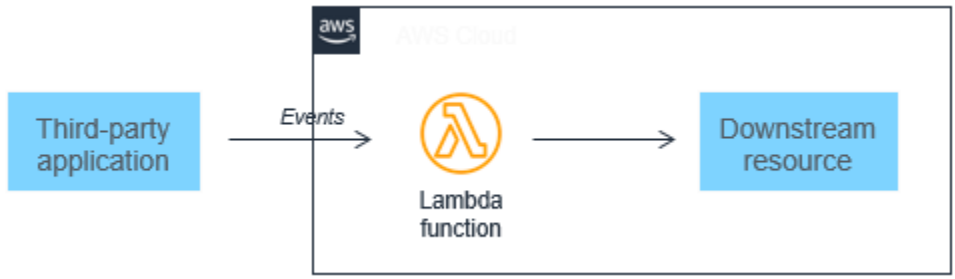
Les deux méthodes contribuent aux avantages des architectures guidées par les événements, comme décrit ci-dessous.

Remplacement des interrogations et des webhooks par des événements

De nombreuses architectures traditionnelles utilisent des mécanismes d'interrogation et de webhook pour communiquer l'état entre différents composants. L'interrogation peut s'avérer très inefficace pour récupérer les mises à jour, car il existe un décalage entre la mise à disposition des nouvelles données et la synchronisation avec les services en aval. Les webhooks ne sont pas toujours pris en charge par les autres microservices auxquels vous souhaitez vous intégrer. Ils peuvent également nécessiter des configurations d'autorisation et d'authentification personnalisées. Dans les deux cas, il est difficile de mettre à l'échelle ces méthodes d'intégration à la demande sans travail supplémentaire de la part des équipes de développement.



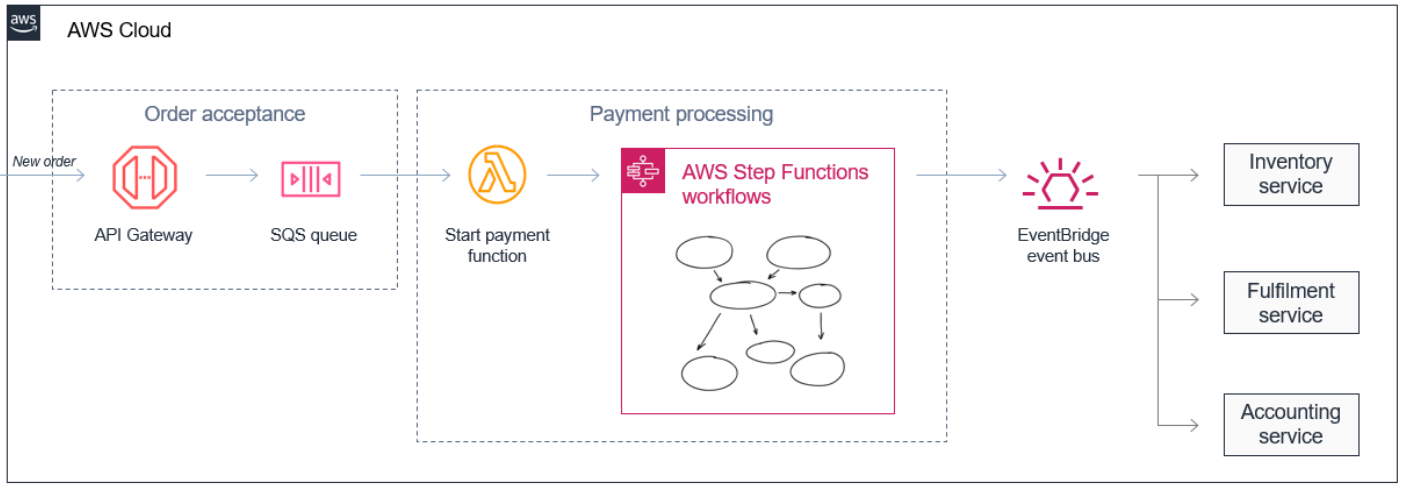
Ces deux mécanismes peuvent être remplacés par des événements, qui peuvent être filtrés, acheminés et redirigés en aval vers les microservices consommateurs. Cette approche peut entraîner une réduction de la consommation de bande passante, de l'utilisation du processeur et potentiellement une baisse des coûts. Ces architectures peuvent également réduire la complexité, étant donné que chaque unité fonctionnelle est plus petite et qu'il y a souvent moins de code.



Les architectures axées sur les événements peuvent également faciliter la conception de near-real-time systèmes, aidant ainsi les entreprises à s'éloigner du traitement par lots. Les événements sont générés au moment où l'état de l'application change. Le code personnalisé d'un microservice doit donc être conçu pour gérer le traitement d'un seul événement. La mise à l'échelle étant gérée par le service Lambda, cette architecture peut gérer des augmentations significatives du trafic sans modifier le code personnalisé. Au fur et à mesure que les événements augmentent verticalement, la couche de calcul qui les traite augmente également.

Réduction de la complexité

Les microservices permettent aux développeurs et aux architectes de simplifier les flux de travail complexes. Par exemple, un monolithe de commerce électronique peut être divisé en processus d'acceptation des commandes et de paiement avec des services d'inventaire, d'exécution et de comptabilité distincts. Ce qui peut s'avérer complexe à gérer et à orchestrer dans un monolithe devient une série de services découplés qui communiquent de manière asynchrone avec les événements.



Cette approche permet également d'assembler des services qui traitent les données à des rythmes différents. Dans ce cas, un microservice d'acceptation des commandes peut stocker de gros volumes de commandes entrantes en mettant les messages en mémoire tampon dans une file d'attente Amazon SQS.

Un service de traitement des paiements, généralement plus lent en raison de la complexité du traitement des paiements, peut recevoir un flux constant de messages provenant de la file d'attente Amazon SQS. Il peut orchestrer une logique complexe de gestion des nouvelles tentatives et des erreurs en utilisant AWS Step Functions et en coordonnant les flux de paiement actifs pour des centaines de milliers de commandes.

Amélioration de la capacité de mise à l’échelle et de l’extensibilité

Les microservices génèrent des événements qui sont généralement publiés sur des services de messagerie tels qu’Amazon SNS et Amazon SQS. Ils se comportent comme une mémoire tampon élastique entre les microservices et aident à gérer la mise à l’échelle lorsque le trafic augmente. Des services tels qu'Amazon EventBridge peuvent ensuite filtrer et acheminer les messages en fonction du contenu de l'événement, tel que défini dans les règles. Par conséquent, les applications basées sur les événements peuvent être plus évolutives et offrir une meilleure redondance que les applications monolithiques.

Ce système est également très extensible, ce qui permet aux autres équipes d’étendre les fonctionnalités et d’en ajouter sans affecter les microservices de traitement des commandes et des paiements. En publiant des événements à l'aide de cette application EventBridge, elle s'intègre aux systèmes existants, tels que le microservice d'inventaire, mais permet également à toute future application de s'intégrer en tant que consommateur d'événements. Les producteurs d’événements ne connaissent pas les consommateurs d’événements, ce qui peut contribuer à simplifier la logique des microservices.

Compromis des architectures guidées par les événements

Latence variable

Contrairement aux applications monolithiques, qui peuvent tout traiter dans le même espace mémoire sur un seul appareil, les applications pilotées par les événements communiquent sur les réseaux. Cette conception introduit une latence variable. Bien qu’il soit possible de concevoir des applications pour réduire la latence, les applications monolithiques peuvent presque toujours être optimisées pour réduire la latence au détriment de la capacité de mise à l’échelle et de la disponibilité.

Les charges de travail qui nécessitent des performances constantes à faible latence, telles que les applications de négociation à haute fréquence dans les banques ou l’automatisation robotique en moins d’une milliseconde dans les entrepôts, ne sont pas de bonnes candidates pour une architecture guidée par les événements.

Cohérence à terme

Un événement représente un changement d’état, et comme de nombreux événements transitent par différents services d’une architecture à un moment donné, ces charges de travail sont souvent cohérentes à terme. Cela complique le traitement des transactions, la gestion des doublons ou la détermination de l’état global exact d’un système.

Certaines charges de travail contiennent une combinaison d'exigences qui sont cohérentes à terme (par exemple, le nombre total de commandes dans l'heure en cours) ou fortement cohérentes (par exemple, l'inventaire actuel). Pour les charges de travail nécessitant une forte cohérence des données, il existe des modèles d'architecture adaptés. Par exemple :

- DynamoDB peut fournir des lectures fortement cohérentes, parfois avec une latence plus élevée, consommant un débit supérieur à celui du mode par défaut. DynamoDB peut également prendre en charge les transactions afin de garantir la cohérence des données.
- Vous pouvez utiliser Amazon RDS pour les fonctionnalités nécessitant des propriétés ACID, bien que les bases de données relationnelles aient généralement une moins bonne capacité de mise à l'échelle qu'une base de données NoSQL comme DynamoDB. Le Proxy Amazon RDS peut aider à gérer le regroupement et la mise à l'échelle des connexions à partir de clients éphémères tels que les fonctions Lambda.

Les architectures basées sur les événements sont généralement conçues en fonction d'événements individuels plutôt que de grands lots de données. En général, les flux de travail sont conçus pour gérer les étapes d'un événement ou d'un flux d'exécution individuel au lieu de traiter plusieurs événements simultanément. En mode sans serveur, le traitement des événements en temps réel est préféré au traitement par lots : les lots doivent être remplacés par de nombreuses mises à jour incrémentales plus petites. Cela peut rendre les charges de travail plus disponibles et évolutives, mais il est également plus difficile pour les événements d'avoir connaissance des autres événements.

Renvoi des valeurs aux appelants

Dans de nombreux cas, les applications basées sur des événements sont asynchrones. Cela signifie que les services des appelants n'attendent pas les requêtes des autres services pour poursuivre d'autres tâches. Il s'agit d'une caractéristique fondamentale des architectures guidées par les événements qui permet la capacité de mise à l'échelle et la flexibilité. Cela signifie que la transmission des valeurs de retour ou du résultat d'un flux de travail est plus complexe que dans les flux d'exécution synchrones.

La plupart des invocations Lambda dans les systèmes de production sont asynchrones et répondent à des événements provenant de services tels qu'Amazon S3 ou Amazon SQS. Dans ces cas, le succès ou l'échec du traitement d'un événement est souvent plus important que le renvoi d'une valeur. Des fonctionnalités telles que les files d'attente de lettres mortes (DLQs) dans Lambda sont fournies pour vous permettre d'identifier et de réessayer les événements ayant échoué, sans avoir à en informer l'appelant.

Débogage entre les services et les fonctions

Le débogage de systèmes guidés par des événements est également différent par rapport à une application monolithique. Lorsque différents systèmes et services transmettent des événements, il est impossible d'enregistrer et de reproduire l'état exact de plusieurs services en cas d'erreur. Étant donné que chaque invocation de service et de fonction comporte des fichiers journaux distincts, il peut être plus compliqué de déterminer ce qu'il est advenu d'un événement spécifique à l'origine d'une erreur.

Trois conditions importantes sont requises pour élaborer une approche de débogage réussie dans les systèmes guidés par les événements. Tout d'abord, un système de journalisation robuste est essentiel, fourni par l'ensemble AWS des services et intégré aux fonctions Lambda par Amazon. CloudWatch Deuxièmement, dans ces systèmes, il est important de s'assurer que chaque événement possède un identifiant de transaction journalisé à chaque étape de la transaction, afin de faciliter la recherche de journaux.

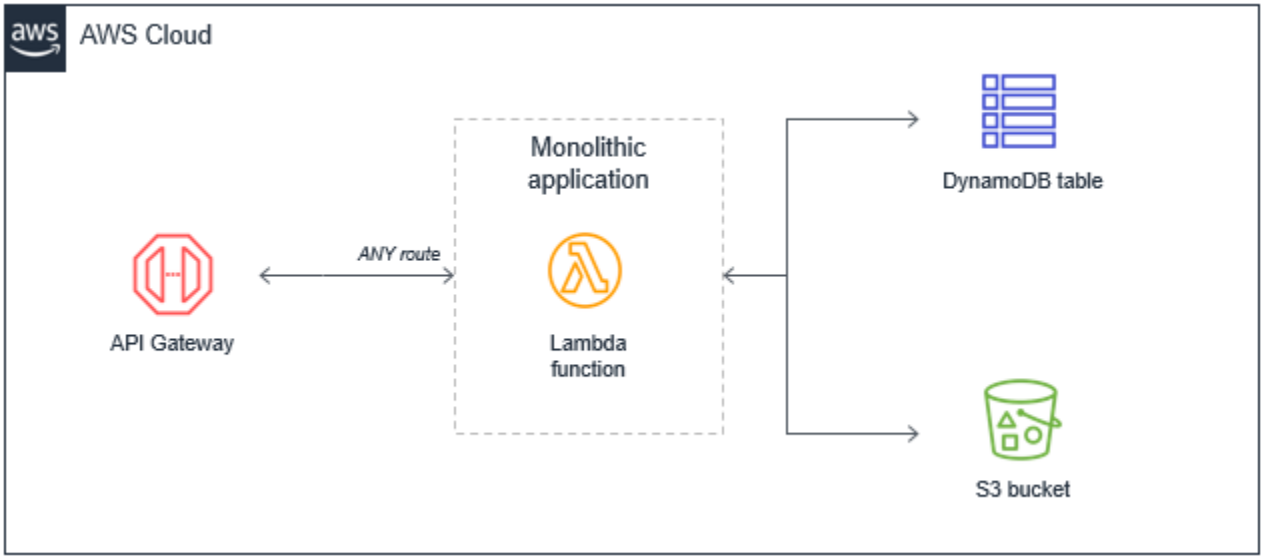
Enfin, il est fortement recommandé d'automatiser l'analyse des journaux à l'aide d'un service de débogage et de surveillance tel qu' AWS X-Ray. Cela peut consommer les journaux de plusieurs invocations et services Lambda, ce qui permet d'identifier plus facilement la cause racine des problèmes. Consultez la Procédure de dépannage pour une présentation détaillée de l'utilisation de X-Ray à des fins de résolution des problèmes.

Anti-modèles dans les applications pilotées par les événements basées sur Lambda

Lorsque vous créez des architectures pilotées par des événements avec Lambda, évitez les anti-modèles courants suivants. Ces modèles fonctionnent mais peuvent augmenter les coûts et la complexité.

Le monolithe Lambda

Dans de nombreuses applications migrées depuis des serveurs traditionnels, telles que les EC2 instances Amazon ou les applications Elastic Beanstalk, les développeurs « modifient et modifient » le code existant. Cela se traduit souvent par une seule fonction Lambda qui contient toute la logique d'application déclenchée pour tous les événements. Pour une application Web de base, une fonction Lambda monolithique gérerait toutes les routes API Gateway et s'intégrerait à toutes les ressources en aval nécessaires.

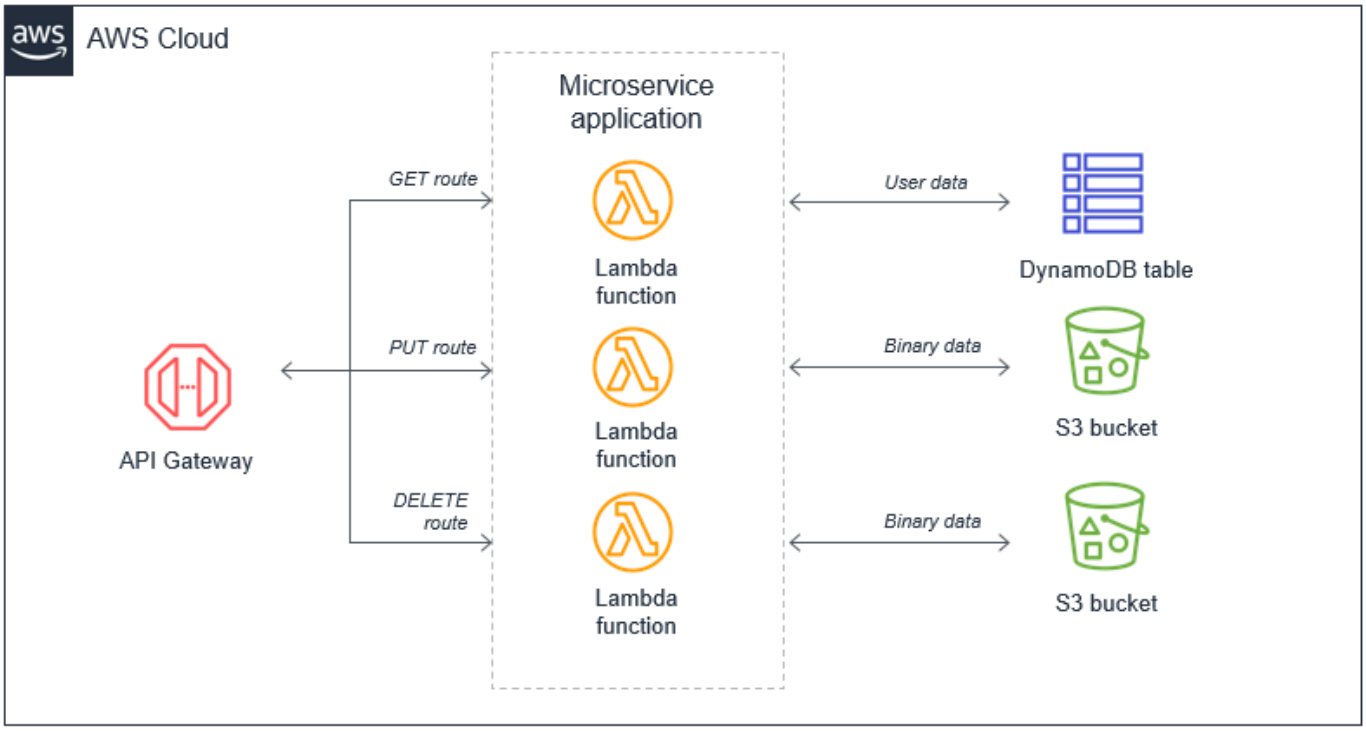


Cette approche présente plusieurs inconvénients :

- Taille du package — La fonction Lambda peut être beaucoup plus importante car elle contient tout le code possible pour tous les chemins, ce qui ralentit l'exécution du service Lambda.
- Moindre privilège difficile à appliquer : le rôle d'exécution de la fonction doit accorder des autorisations à toutes les ressources nécessaires pour tous les chemins, ce qui rend les autorisations très étendues. Il s'agit d'un problème de sécurité. De nombreux chemins du monolithe fonctionnel n'ont pas besoin de toutes les autorisations accordées.
- Mise à niveau plus difficile : dans un système de production, toute mise à niveau d'une seule fonction est plus risquée et peut entraîner l'arrêt du fonctionnement de l'application dans son ensemble. La mise à niveau d'un seul chemin dans la fonction Lambda est une mise à niveau de l'ensemble de la fonction.
- Maintenance plus difficile : il est plus difficile d'avoir plusieurs développeurs travaillant sur le service puisqu'il s'agit d'un référentiel de code monolithique. Cela alourdit également la charge cognitive des développeurs et complique la création d'une couverture de test appropriée pour le code.
- Réutilisation du code plus difficile : il est plus difficile de séparer les bibliothèques réutilisables des monolithes, ce qui complique la réutilisation du code. Au fur et à mesure que vous développez et soutenez de plus en plus de projets, il peut être plus difficile de soutenir le code et de mettre à l'échelle la vitesse de votre équipe.
- Tests plus difficiles : à mesure que le nombre de lignes de code augmente, il devient plus difficile d'effectuer des tests unitaires pour toutes les combinaisons possibles d'entrées et de points

d’entrée dans la base de code. Il est généralement plus facile de mettre en œuvre des tests unitaires pour les petits services avec moins de code.

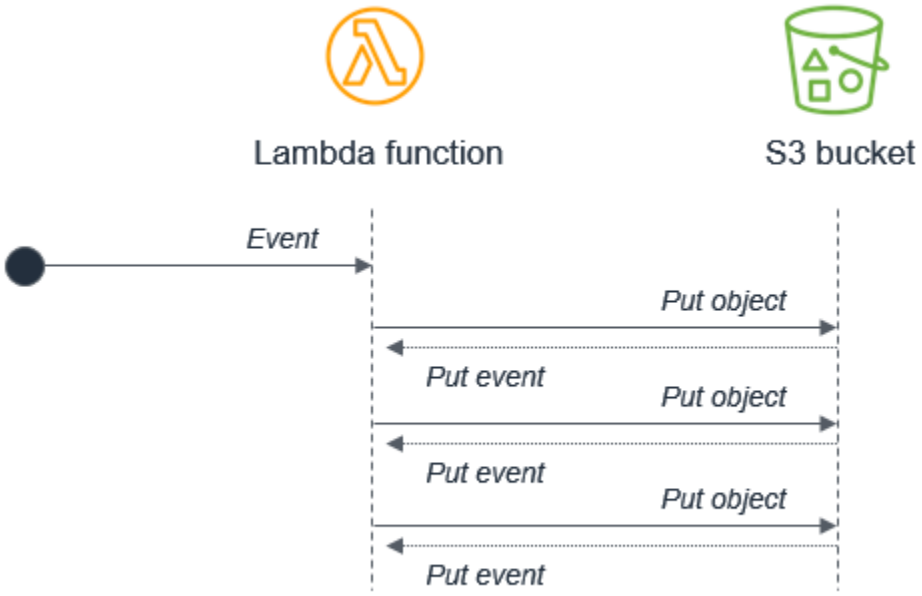
L'alternative préférée consiste à décomposer la fonction Lambda monolithique en microservices individuels, en mappant une seule fonction Lambda à une seule tâche bien définie. Dans cette application Web simple dotée de quelques points de terminaison d’API, l’architecture basée sur les microservices qui en résulte peut être basée sur les routes API Gateway.



Modèles récursifs qui provoquent des fonctions Lambda incontrôlables

AWS les services génèrent des événements qui invoquent des fonctions Lambda, et les fonctions Lambda peuvent envoyer des messages aux services. AWS En général, le service ou la ressource qui invoque une fonction Lambda doit être différent du service ou de la ressource vers lequel la fonction émet des sorties. Si vous ne parvenez pas à gérer cela, vous risquez de créer des boucles infinies.

Par exemple, une fonction Lambda écrit un objet dans un objet Amazon S3, qui à son tour invoque la même fonction Lambda via un événement put. L’invocation entraîne l’écriture d’un deuxième objet dans le compartiment, qui invoque la même fonction Lambda :



Bien que le potentiel de boucles infinies existe dans la plupart des langages de programmation, cet anti-modèle est susceptible de consommer davantage de ressources dans les applications sans serveur. Lambda et Amazon S3 évoluent automatiquement en fonction du trafic, de sorte que la boucle peut amener Lambda à évoluer pour utiliser toute la simultanéité disponible et Amazon S3 continuera à écrire des objets et à générer davantage d'événements pour Lambda.

Cet exemple utilise S3, mais le risque de boucles récursives existe également dans Amazon SNS, Amazon SQS, DynamoDB et d'autres services. Vous pouvez utiliser la détection de boucle récursive pour détecter et éviter cet anti-modèle.

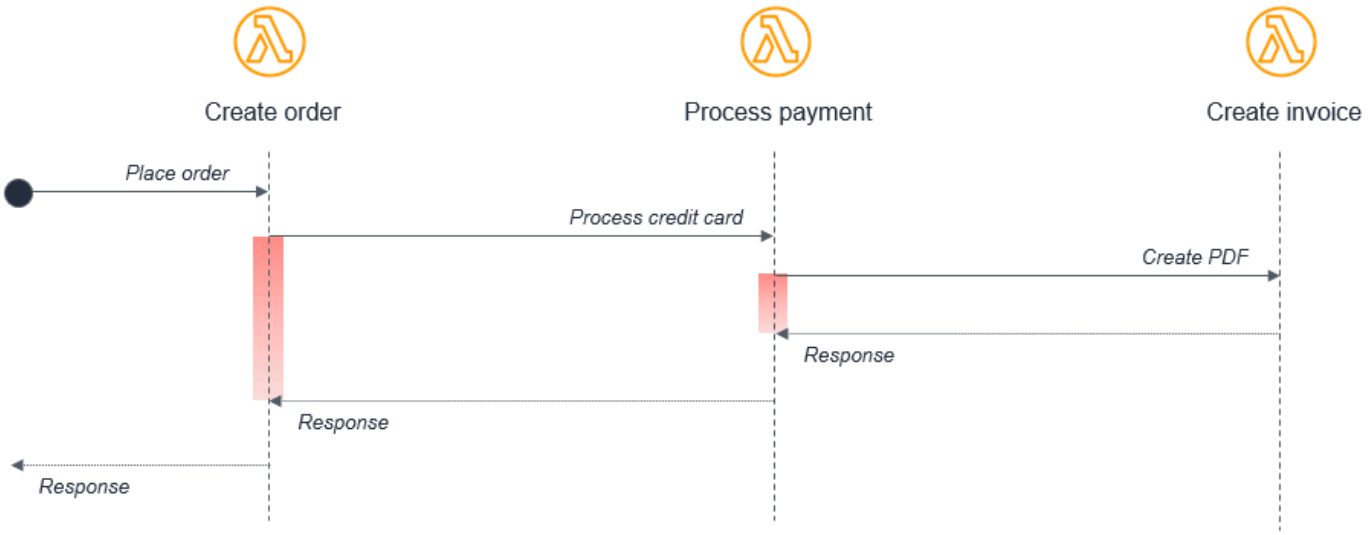
Fonctions Lambda appelant des fonctions Lambda

Les fonctions permettent l'encapsulation et la réutilisation du code. La plupart des langages de programmation prennent en charge le concept d'appel synchrone de fonctions au sein d'une base de code. Dans ce cas, l'appelant attend la réponse de la fonction.

> **Note**
>
> Bien que les fonctions Lambda appelant directement d'autres fonctions Lambda constituent généralement un anti-modèle en raison de problèmes de coût et de complexité, cela ne s'applique pas aux fonctions durables, qui sont spécifiquement conçues pour orchestrer des flux de travail en plusieurs étapes en invoquant d'autres fonctions.

Lorsque cela se produit sur un serveur traditionnel ou une instance virtuelle, le planificateur du système d'exploitation passe à d'autres tâches disponibles. Le fait que le processeur fonctionne à 0 % ou à 100 % n'a aucune incidence sur le coût global de l'application, puisque vous payez le coût fixe lié à la possession et à l'exploitation d'un serveur.

Ce modèle ne s'adapte souvent pas bien au développement sans serveur. Prenons l'exemple d'une simple application de commerce électronique composée de trois fonctions Lambda qui traitent une commande :



Dans ce cas, la fonction Créer une commande appelle la fonction Traiter le paiement, qui à son tour appelle la fonction Créer une facture. Bien que ce flux synchrone puisse fonctionner au sein d'une seule application sur un serveur, il introduit plusieurs problèmes évitables dans une architecture sans serveur distribuée :

- Coût : avec Lambda, vous payez pour la durée d'une invocation. Dans cet exemple, pendant que les fonctions Créer une facture s'exécutent, deux autres fonctions sont également exécutées en attente, comme indiqué en rouge sur le diagramme.
- Gestion des erreurs : dans les invocations imbriquées, la gestion des erreurs peut devenir beaucoup plus complexe. Par exemple, une erreur dans Créer une facture peut nécessiter que la fonction de traitement du paiement annule le débit, ou qu'elle réessaie le processus de création de facture.
- Couplage fort : le traitement d'un paiement prend généralement plus de temps que la création d'une facture. Dans ce modèle, la disponibilité de l'ensemble du flux de travail est limitée par la fonction la plus lente.

- Mise à l'échelle : la simultanéité des trois fonctions doit être égale. Dans un système occupé, cela utilise plus de simultanéité que ce qui serait nécessaire autrement.

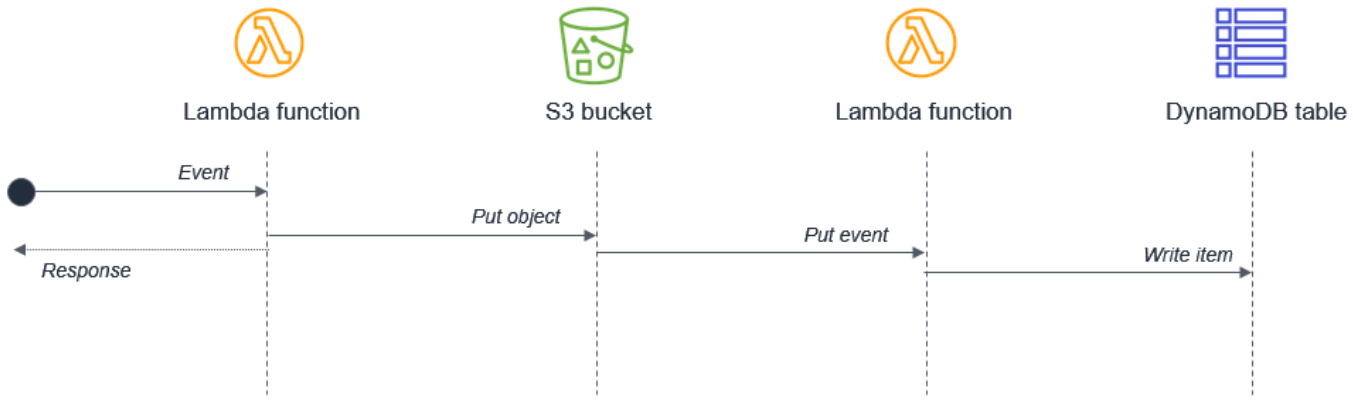
Dans les applications sans serveur, il existe deux approches courantes pour éviter ce schéma. Tout d'abord, utilisez une file d'attente Amazon SQS entre les fonctions Lambda. Si un processus en aval est plus lent qu'un processus en amont, la file d'attente conserve les messages de manière durable et dissocie les deux fonctions. Dans cet exemple, la fonction Create order publierait un message dans une file d'attente Amazon SQS, et la fonction Process payment consomme les messages de la file d'attente.

La deuxième approche consiste à utiliser AWS Step Functions. Pour les processus complexes présentant plusieurs types d'échec et une logique de nouvelle tentative, Step Functions peut aider à réduire la quantité de code personnalisé nécessaire pour orchestrer le flux de travail. Par conséquent, Step Functions orchestre les tâches et gère efficacement les erreurs et les nouvelles tentatives, et les fonctions Lambda ne contiennent que de la logique métier.

Attente synchrone au sein d'une seule fonction Lambda

Assurez-vous que les activités potentiellement simultanées ne sont pas planifiées de manière synchrone au sein d'une seule fonction Lambda. Par exemple, une fonction Lambda peut écrire dans un compartiment S3, puis dans une table DynamoDB :

Dans cette conception, les temps d'attente sont cumulés, car les activités sont séquentielles. Dans les cas où la deuxième tâche dépend de l'achèvement de la première, vous pouvez peut-être réduire le temps d'attente total et le coût d'exécution en ayant deux fonctions Lambda distinctes :



Dans cette conception, la première fonction Lambda répond immédiatement après avoir placé l'objet dans le compartiment Amazon S3. Le service S3 invoque la deuxième fonction Lambda, qui écrit ensuite des données dans la table DynamoDB. Cette approche réduit le temps d'attente total lors des exécutions de fonctions Lambda.

Conception d’applications Lambda

Une application événementielle bien conçue utilise une combinaison de AWS services et de code personnalisé pour traiter et gérer les demandes et les données. Ce chapitre se concentre sur les sujets spécifiques à Lambda en matière de conception d’applications. Les architectes sans serveur doivent tenir compte de nombreuses considérations importantes lorsqu’ils conçoivent des applications pour des systèmes de production très sollicités.

Bon nombre des pratiques exemplaires applicables au développement de logiciels et aux systèmes distribués s’appliquent également au développement d’applications sans serveur. L’objectif global est de développer des charges de travail qui sont :

- Fiables : offrez à vos utilisateurs finaux un haut niveau de disponibilité. Les services AWS sans serveur sont fiables, car ils sont également conçus pour résister aux défaillances.
- Durables : offrez des options de stockage qui répondent aux besoins de durabilité de votre charge de travail.
- Sécurisées : suivez les pratiques exemplaires et utilisez les outils fournis pour sécuriser l’accès aux charges de travail et limiter le rayon d’impact.
- Performantes : utilisez les ressources informatiques avec efficacité et répondez aux besoins de performance de vos utilisateurs finaux.
- Rentables : concevez des architectures qui évitent les coûts inutiles, qui peuvent se mettre à l’échelle sans dépenses excessives, et également être mises hors service sans frais généraux importants.

Les principes de conception suivants peuvent vous aider à créer des charges de travail répondant à ces objectifs. Tous les principes ne s’appliquent pas à toutes les architectures, mais ils devraient vous guider dans vos décisions générales en matière d’architecture.

Rubriques

- Utilisation de services plutôt que de code personnalisé
- Présentation des niveaux d’abstraction Lambda
- Implémenter l’absence d’état dans les fonctions
- Minimiser le couplage
- Générer des données à la demande plutôt que pour des lots
- Pensez à AWS Step Functions l'orchestration

- Implémenter l'idempotence
- Utiliser plusieurs AWS comptes pour gérer les quotas

Utilisation de services plutôt que de code personnalisé

Les applications sans serveur comprennent généralement plusieurs AWS services, intégrés à du code personnalisé exécuté dans les fonctions Lambda. Lambda peut être intégré à la plupart des AWS services, mais les services les plus couramment utilisés dans les applications sans serveur sont les suivants :

Catégorie	AWS service
Calcul	AWS Lambda
Stockage de données	Amazon S3 Amazon DynamoDB Amazon RDS
API	Amazon API Gateway
Intégration d'applications	Amazon EventBridge Amazon SNS Amazon SQS
Orchestration	AWS Step Functions
Données de streaming et analytique	Amazon Data Firehose

> **Note**
>
> De nombreux services sans serveur fournissent la réplication et la prise en charge de plusieurs régions, notamment DynamoDB et Amazon S3. Les fonctions Lambda peuvent être déployées dans plusieurs régions dans le cadre d'un pipeline de déploiement, et API

Gateway peut être configurée pour prendre en charge cette configuration. Consultez cet exemple d'architecture qui montre comment cela peut être réalisé.

Il existe de nombreux modèles courants bien établis dans les architectures distribuées que vous pouvez créer vous-même ou implémenter à l'aide de AWS services. Pour la plupart des clients, le développement de ces modèles à partir de zéro présente peu d'intérêt commercial. Lorsque votre application a besoin de l'un de ces modèles, utilisez le AWS service correspondant :

Modèle	AWS service
File d'attente	Amazon SQS
Bus d'événement	Amazon EventBridge
Publication-abonnement (diffusion en éventail)	Amazon SNS
Orchestration	AWS Step Functions
API	Amazon API Gateway
Flux d'événement	Amazon Kinesis

Ces services sont conçus pour s'intégrer à Lambda et vous pouvez utiliser l'infrastructure en tant que code (IaC) pour créer et supprimer des ressources dans les services. Vous pouvez utiliser n'importe lequel de ces services via le kit SDK AWS sans avoir à installer d'applications ou à configurer des serveurs. Maîtriser l'utilisation de ces services via le code dans vos fonctions Lambda est une étape importante pour produire des applications sans serveur bien conçues.

Présentation des niveaux d'abstraction Lambda

Le service Lambda limite votre accès aux systèmes d'exploitation, aux hyperviseurs et au matériel sous-jacents qui exécutent vos fonctions Lambda. Le service améliore et modifie continuellement l'infrastructure afin d'ajouter des fonctionnalités, de réduire les coûts et de rendre le service plus performant. Votre code ne doit émettre aucune hypothèse quant à l'architecture de Lambda et à l'affinité matérielle.

De même, les intégrations de Lambda avec d'autres services sont gérées par AWS, et seul un petit nombre d'options de configuration vous sont proposées. Par exemple, quand API Gateway et

Lambda interagissent, il n'y a aucun concept d'équilibrage de charge, puisqu'il est entièrement géré par les services. Vous n'avez également aucun contrôle direct sur les zones de disponibilité utilisées par les services lorsqu'ils invoquent des fonctions à tout moment, ni sur la manière dont Lambda détermine quand réduire ou augmenter verticalement le nombre d'environnements d'exécution.

Cette abstraction vous aide à vous concentrer sur les aspects d'intégration de votre application, le flux de données et la logique métier dans laquelle votre charge de travail apporte de la valeur à vos utilisateurs finaux. En permettant aux services de gérer les mécanismes sous-jacents, vous pouvez développer des applications plus rapidement avec moins de code personnalisé à gérer.

Implémenter l'absence d'état dans les fonctions

Pour les fonctions Lambda standard, vous devez partir du principe que l'environnement n'existe que pour un seul appel. La fonction doit initialiser tout état requis lors de son premier démarrage. Par exemple, votre fonction peut nécessiter de récupérer des données dans une table DynamoDB. Elle doit valider toute modification permanente des données sur un magasin durable tel qu'Amazon S3, DynamoDB ou Amazon SQS avant de quitter. Elle ne doit pas s'appuyer sur des structures de données ou des fichiers temporaires existants, ni sur un état interne qui serait géré par de multiples invocations.

Lorsque vous utilisez Durable Functions, l'état est automatiquement préservé entre les appels, ce qui élimine le besoin de le conserver manuellement sur le stockage externe. Cependant, vous devez toujours suivre les principes de l'apatridie pour toutes les données qui ne sont pas explicitement gérées par le biais du DurableContext.

Pour initialiser les connexions aux bases de données et les bibliothèques, ou pour initialiser l'état de chargement, vous pouvez tirer parti de l'initialisation statique. Les environnements d'exécution étant réutilisés dans la mesure du possible pour améliorer les performances, vous pouvez amortir le temps nécessaire à l'initialisation de ces ressources sur plusieurs invocations. Cependant, vous ne devez stocker aucune variable ou donnée utilisée dans la fonction dans cette portée globale.

Minimiser le couplage

La plupart des architectures devraient préférer les fonctions plus nombreuses et plus petites aux fonctions moins nombreuses et plus grandes. Le but de chaque fonction doit être de gérer l'événement transmis à la fonction, sans aucune connaissance ni attente quant au flux de travail global ou au volume des transactions. Cela rend la fonction indépendante de la source de l'événement avec un couplage minimal avec les autres services.

Toutes les constantes de portée globale qui changent rarement doivent être implémentées en tant que variables d'environnement afin de permettre les mises à jour sans déploiement. Tous les secrets ou informations sensibles doivent être stockés dans AWS Systems Manager Parameter Store ou dans AWS Secrets Manager et chargés par la fonction. Comme ces ressources sont propres à un compte, vous pouvez créer des pipelines de génération sur plusieurs comptes. Les pipelines chargent les secrets appropriés par environnement, sans les exposer aux développeurs ni nécessiter de modification du code.

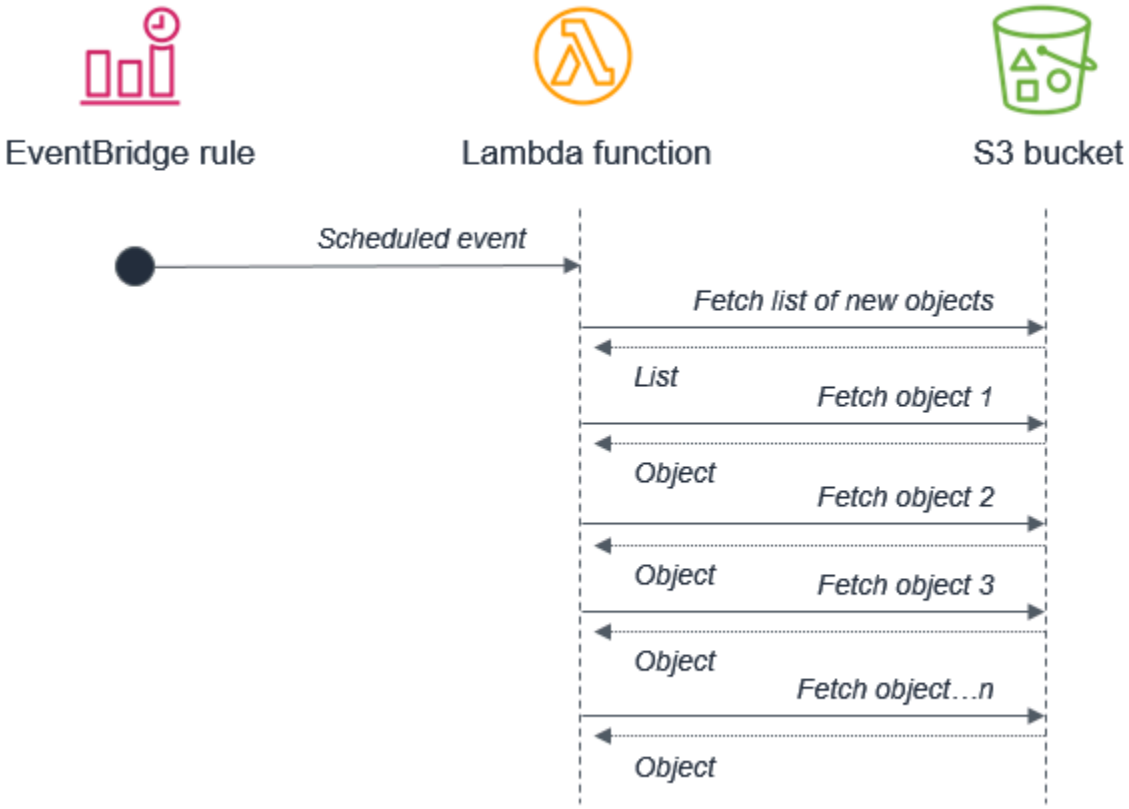
Générer des données à la demande plutôt que pour des lots

De nombreux systèmes traditionnels sont conçus pour fonctionner périodiquement et traiter des lots de transactions accumulés au fil du temps. Par exemple, une application bancaire peut s'exécuter toutes les heures pour traiter les transactions des guichets automatiques dans des registres centraux. Dans les applications basées sur Lambda, le traitement personnalisé doit être déclenché par chaque événement, ce qui permet au service d'augmenter verticalement la simultanéité selon les besoins, afin de fournir un traitement des transactions en temps quasi réel.

Alors que les fonctions Lambda standard sont limitées à 15 minutes d'exécution, les fonctions durables peuvent fonctionner jusqu'à un an, ce qui les rend adaptées aux besoins de traitement de longue durée. Cependant, vous devez toujours préférer le traitement piloté par les événements au traitement par lots lorsque cela est possible.

Bien que vous puissiez exécuter des tâches cron dans des applications sans serveur en utilisant des expressions planifiées pour les règles dans Amazon EventBridge, celles-ci doivent être utilisées avec parcimonie ou en dernier recours. Dans toute tâche planifiée qui traite un lot, il est possible que le volume de transactions augmente au-delà de ce qui peut être traité dans la limite de durée Lambda de 15 minutes. Si les limites des systèmes externes vous obligent à utiliser un planificateur, vous devez généralement planifier pour la période récurrente la plus courte possible.

Par exemple, il n'est pas recommandé d'utiliser un traitement par lots qui déclenche une fonction Lambda pour récupérer une liste de nouveaux objets Amazon S3. Cela est dû au fait que le service peut recevoir plus de nouveaux objets entre les lots que ce qui peut être traité dans le cadre d'une fonction Lambda de 15 minutes.



Amazon S3 doit plutôt invoquer la fonction Lambda chaque fois qu'un nouvel objet est placé dans le compartiment. Cette démarche est bien plus facile à mettre à l'échelle et fonctionne en temps quasi réel.

Pensez à AWS Step Functions l'orchestration

Les flux de travail qui impliquent une logique de ramification, différents types de modèles d'échec et une logique de nouvelle tentative utilisent généralement un orchestrateur pour suivre l'état de l'exécution globale. Évitez d'utiliser les fonctions Lambda à cette fin, car cela se traduit par un couplage fort et par un routage de gestion du code complexe.

Avec AWS Step Functions, vous utilisez des machines d'état pour gérer l'orchestration. Cela extrait la logique de gestion des erreurs, de routage et de branchement de votre code, en la remplaçant par

des machines d'état déclarées à l'aide de JSON. En plus de rendre les flux de travail plus robustes et observables, vous pouvez également ajouter la gestion des versions aux flux de travail et faire de la machine d'état une ressource codifiée que vous pouvez ajouter à un référentiel de code.

Il est courant que les flux de travail simples dans les fonctions Lambda se complexifient au fil du temps. Lorsque vous utilisez une application de production sans serveur, il est important d'identifier le moment où cela se produit, afin de pouvoir migrer cette logique vers une machine d'état.

Implémenter l'idempotence

AWS les services sans serveur, y compris Lambda, sont tolérants aux pannes et conçus pour gérer les défaillances. Par exemple, si un service invoque une fonction Lambda et qu'il y a une interruption de service, Lambda invoque votre fonction dans une autre zone de disponibilité. Si la fonction renvoie une erreur, Lambda relance l'invocation.

Comme le même événement peut être reçu plusieurs fois, les fonctions doivent être conçues pour être idempotentes. Cela signifie que le fait de recevoir le même événement plusieurs fois ne change pas le résultat au-delà de la première réception de l'événement.

Vous pouvez implémenter l'idempotence dans les fonctions Lambda en utilisant une table DynamoDB pour suivre les identifiants récemment traités afin de déterminer si la transaction a déjà été traitée. La table DynamoDB implémente généralement une valeur de durée de vie (TTL) pour faire expirer les éléments afin de limiter l'espace de stockage utilisé.

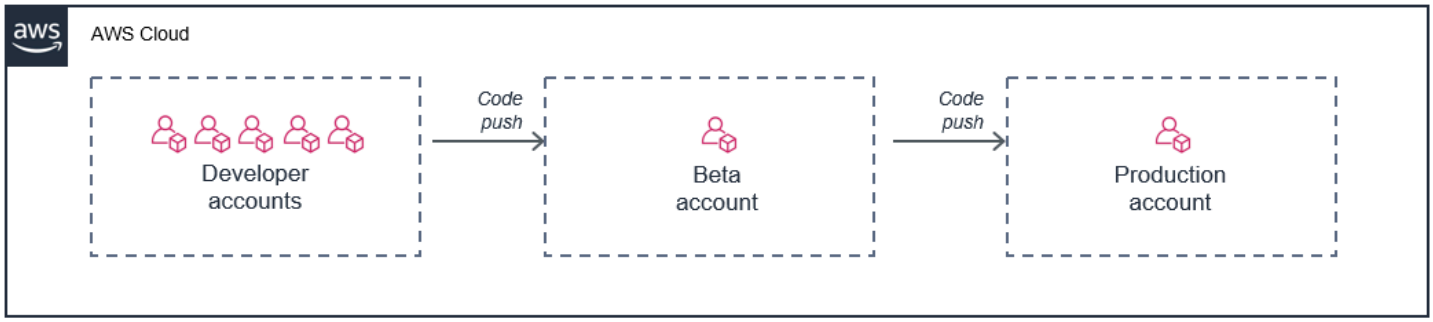
Utiliser plusieurs AWS comptes pour gérer les quotas

De nombreux quotas de service AWS sont définis au niveau du compte. Cela signifie qu'à mesure que vous ajoutez de nouvelles charges de travail, vous pouvez rapidement épuiser vos limites.

Un moyen efficace de résoudre ce problème consiste à utiliser plusieurs AWS comptes, en consacrant chaque charge de travail à son propre compte. Cela empêche le partage des quotas avec d'autres charges de travail ou des ressources non liées à la production.

Par ailleurs, grâce à AWS Organizations, il est possible de gérer de manière centralisée la facturation, la conformité et la sécurité de ces comptes. Vous pouvez associer des politiques à des groupes de comptes pour éviter les scripts personnalisés et les processus manuels.

Une approche courante consiste à fournir un AWS compte à chaque développeur, puis à utiliser des comptes distincts pour la phase de déploiement et de production de la version bêta :



Dans ce modèle, chaque développeur a son propre ensemble de limites pour le compte, de sorte que leur utilisation n'a aucune incidence sur votre environnement de production. Cette approche permet également aux développeurs de tester les fonctions Lambda localement sur leurs machines de développement par rapport aux ressources cloud actives de leurs comptes individuels.

Création de votre première fonction Lambda

Pour démarrer avec Lambda, utilisez la console Lambda pour créer une fonction. En quelques minutes, vous pouvez créer et déployer une fonction, mais aussi la tester dans la console.

Au cours du didacticiel, vous apprendrez certains concepts fondamentaux de Lambda, tels que la façon de transmettre des arguments à votre fonction à l'aide de l'objet d'événement Lambda. Vous apprendrez également comment renvoyer les résultats des journaux de votre fonction et comment consulter les journaux d'invocation de votre fonction dans Amazon CloudWatch Logs.

Pour simplifier les choses, vous devez créer votre fonction à l'aide de l'exécution Python ou Node.js. Avec ces langages interprétés, vous pouvez modifier le code de fonction directement dans l'éditeur de code intégré à la console. Avec des langages compilés tels que Java et C#, vous devez créer un package de déploiement sur votre machine de génération locale et le télécharger sur Lambda. Pour en savoir plus sur le déploiement de fonctions sur Lambda à l'aide d'autres environnements d'exécution, consultez les liens figurant dans la section the section called “Étapes suivantes”.

> **Tip**
>
> Pour apprendre à créer des solutions sans serveur, consultez le Guide du développeur sans serveur.

Conditions préalables

Inscrivez-vous pour un Compte AWS

Si vous n'en avez pas Compte AWS, procédez comme suit pour en créer un.

Pour vous inscrire à un Compte AWS

1. Ouvrez l'https://portal.aws.amazon.com/billing/inscription.
2. Suivez les instructions en ligne.

   Dans le cadre de la procédure d’inscription, vous recevrez un appel téléphonique ou un SMS et vous saisirez un code de vérification en utilisant le clavier numérique du téléphone.

   Lorsque vous vous inscrivez à un Compte AWS, un Utilisateur racine d'un compte AWSest créé. Par défaut, seul l’utilisateur racine a accès à l’ensemble des Services AWS et des ressources

de ce compte. La meilleure pratique de sécurité consiste à attribuer un accès administratif à un utilisateur, et à utiliser uniquement l'utilisateur racine pour effectuer les tâches nécessitant un accès utilisateur racine.

AWS vous envoie un e-mail de confirmation une fois le processus d'inscription terminé. À tout moment, vous pouvez consulter l'activité actuelle de votre compte et gérer votre compte en accédant à https://aws.amazon.com/et en choisissant Mon compte.

Création d'un utilisateur doté d'un accès administratif

Après vous être inscrit à un Compte AWS, sécurisez Utilisateur racine d'un compte AWS AWS IAM Identity Center, activez et créez un utilisateur administratif afin de ne pas utiliser l'utilisateur root pour les tâches quotidiennes.

Sécurisez votre Utilisateur racine d'un compte AWS

1. Connectez-vous en AWS Management Consoletant que propriétaire du compte en choisissant Utilisateur root et en saisissant votre adresse Compte AWS e-mail. Sur la page suivante, saisissez votre mot de passe.

   Pour obtenir de l'aide pour vous connecter en utilisant l'utilisateur racine, consultez Connexion en tant qu'utilisateur racine dans le Guide de l'utilisateur Connexion à AWS .
2. Activez l'authentification multifactorielle (MFA) pour votre utilisateur racine.

   Pour obtenir des instructions, voir Activer un périphérique MFA virtuel pour votre utilisateur Compte AWS root (console) dans le guide de l'utilisateur IAM.

Création d'un utilisateur doté d'un accès administratif

1. Activez IAM Identity Center.

   Pour obtenir des instructions, consultez Activation d' AWS IAM Identity Center dans le Guide de l'utilisateur AWS IAM Identity Center .
2. Dans IAM Identity Center, octroyez un accès administratif à un utilisateur.

   Pour un didacticiel sur l'utilisation du Répertoire IAM Identity Center comme source d'identité, voir Configurer l'accès utilisateur par défaut Répertoire IAM Identity Center dans le Guide de AWS IAM Identity Center l'utilisateur.

Connexion en tant qu'utilisateur doté d'un accès administratif

- Pour vous connecter avec votre utilisateur IAM Identity Center, utilisez l'URL de connexion qui a été envoyée à votre adresse e-mail lorsque vous avez créé l'utilisateur IAM Identity Center.

  Pour obtenir de l'aide pour vous connecter en utilisant un utilisateur d'IAM Identity Center, consultez la section Connexion au portail AWS d'accès dans le guide de l'Connexion à AWS utilisateur.

Attribution d'un accès à d'autres utilisateurs

1. Dans IAM Identity Center, créez un ensemble d'autorisations qui respecte la bonne pratique consistant à appliquer les autorisations de moindre privilège.

   Pour obtenir des instructions, consultez Création d'un ensemble d'autorisations dans le Guide de l'utilisateur AWS IAM Identity Center .
2. Attribuez des utilisateurs à un groupe, puis attribuez un accès par authentification unique au groupe.

   Pour obtenir des instructions, consultez Ajout de groupes dans le Guide de l'utilisateur AWS IAM Identity Center .

Créer une fonction Lambda à l'aide de la console

Dans cet exemple, votre fonction prend un objet JSON contenant deux valeurs entières étiquetées `"length"`et `"width"`. La fonction multiplie ces valeurs pour calculer une zone et la renvoie sous forme de chaîne JSON.

Votre fonction imprime également la zone calculée, ainsi que le nom de son groupe de CloudWatch logs. Plus loin dans le didacticiel, vous apprendrez à utiliser les CloudWatch journaux pour afficher les enregistrements de l'invocation de vos fonctions.

Pour créer une fonction Lambda Hello world à l'aide de la console

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez Créer une fonction.
3. Sélectionnez Créer à partir de zéro.
4. Dans le volet Informations de base, pour Nom de la fonction, saisissez **`myLambdaFunction`**.

5. Pour Runtime, choisissez Node.js 24 ou Python 3.14.
6. Laissez Architecture définie sur x86_64 et choisissez Créer une fonction.

En plus d'une simple fonction qui renvoie le message `Hello from Lambda!`, Lambda crée également un rôle d'exécution pour votre fonction. Un rôle d'exécution est un rôle Gestion des identités et des accès AWS (IAM) qui accorde à une fonction Lambda l'autorisation d' Services AWS accès et de ressources. Pour votre fonction, le rôle créé par Lambda accorde des autorisations de base pour CloudWatch écrire dans Logs.

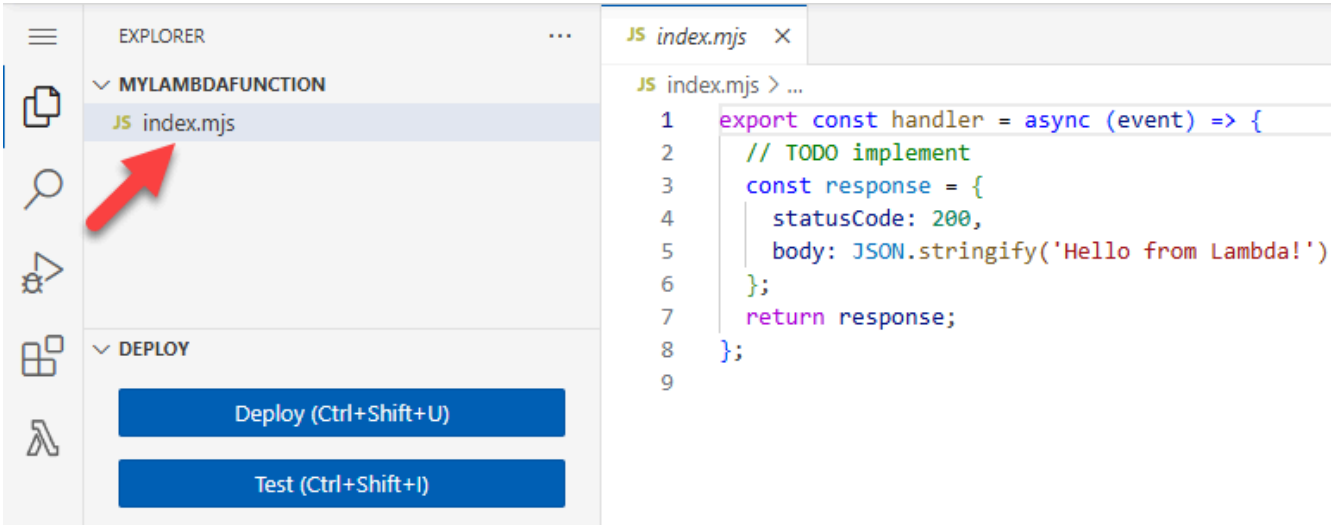
Utilisez l'éditeur de code intégré de la console pour remplacer le code Hello world créé par Lambda avec votre propre code de fonction.

Node.js

Pour modifier le code dans la console

1. Cliquez sur l'onglet Code.

   Dans l'éditeur de code intégré de la console, vous devriez voir le code de fonction créé par Lambda. Si vous ne voyez pas l'onglet index.mjs dans l'éditeur de code, sélectionnez index.mjs dans l'explorateur de fichiers, comme le montre le schéma suivant.

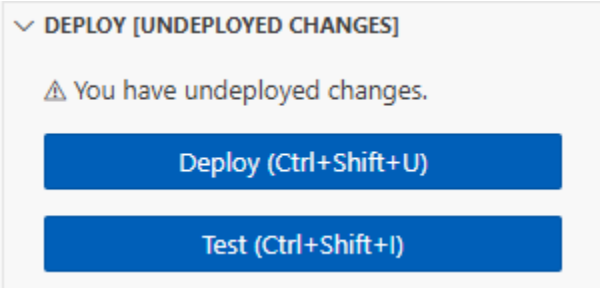


2. Collez le code suivant dans l'onglet index.mjs, en remplaçant le code créé par Lambda.

```
export const handler = async (event, context) => {

  const length = event.length;
  const width = event.width;
```

```
  let area = calculateArea(length, width);
  console.log(`The area is ${area}`);

  console.log('CloudWatch log group: ', context.logGroupName);

  let data = {
    "area": area,
  };
    return JSON.stringify(data);

  function calculateArea(length, width) {
    return length * width;
  }
};
```

3. Dans la section DÉPLOYER, choisissez Déployer pour mettre à jour le code de votre fonction :



Comprendre le code de votre fonction

Avant de passer à l'étape suivante, examinons le code de la fonction pour comprendre certains concepts Lambda clés.

- Le gestionnaire Lambda :

  Votre fonction Lambda contient une fonction Node.js nommée `handler`. Une fonction Lambda en Node.js peut contenir plusieurs fonctions Node.js, mais la fonction gestionnaire est toujours le point d'entrée de votre code. Lorsque votre fonction est invoquée, Lambda exécute cette méthode.

  Lorsque vous avez créé votre fonction Hello world à l'aide de la console, Lambda a automatiquement défini le nom de la méthode du gestionnaire pour votre fonction sur `handler`. Veillez à ne pas modifier le nom de cette fonction Node.js. Dans ce cas, Lambda ne pourra pas exécuter votre code lorsque vous invoquerez votre fonction.

Pour en savoir plus sur le gestionnaire Lambda en Node.js, voir the section called “Handler (Gestionnaire)”.

- L'objet de l'événement Lambda :

  La fonction `handler` accepte deux arguments, `event` et `context`. Un événement dans Lambda est un document au format JSON qui contient des données à traiter par votre fonction.

  Si votre fonction est invoquée par une autre personne Service AWS, l'objet d'événement contient des informations sur l'événement à l'origine de l'invocation. Par exemple, si votre fonction est invoquée lorsqu’un objet est chargé dans un compartiment Amazon Simple Storage Service (Amazon S3), l’événement contient le nom du compartiment et la clé de l’objet.

  Dans cet exemple, vous allez créer un événement dans la console en saisissant un document au format JSON avec deux paires clé-valeur.

- L'objet de contexte Lambda :

  Le deuxième argument que votre fonction accepte est `context`. Lambda transmet automatiquement l'objet de contexte à votre fonction. L'objet de contexte contient des informations supplémentaires sur l'invocation de la fonction et l'environnement d'exécution.

  Vous pouvez utiliser l'objet de contexte pour générer des informations sur l'invocation de votre fonction à des fins de surveillance. Dans cet exemple, votre fonction utilise le `logGroupName` paramètre pour afficher le nom de son groupe de CloudWatch journaux.

  Pour en savoir plus sur l'objet de contexte Lambda en Node.js, voir the section called “Contexte”.

- Connexion à Lambda :

  Avec Node.js, vous pouvez utiliser des méthodes de la console telles que `console.log` et `console.error` pour envoyer des informations au journal de votre fonction. L'exemple de code utilise `console.log` des instructions pour afficher la zone calculée et le nom du groupe CloudWatch Logs de la fonction. Vous pouvez également utiliser n'importe quelle bibliothèque de journalisation qui écrit dans `stdout` ou `stderr`.

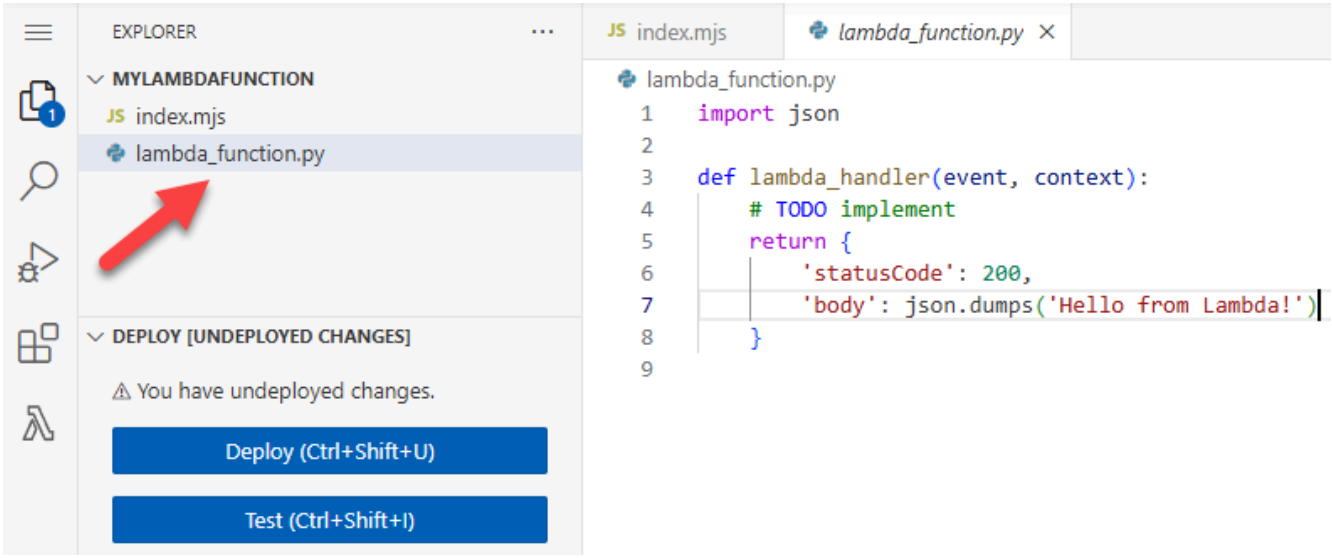
  Pour en savoir plus, veuillez consulter la section the section called “Logging”. Pour en savoir plus sur la connexion à d'autres environnements d'exécution, consultez les pages « Génération avec » correspondant aux environnements d'exécution qui vous intéressent.

Python

Pour modifier le code dans la console

1. Cliquez sur l'onglet Code.

   Dans l'éditeur de code intégré de la console, vous devriez voir le code de fonction créé par Lambda. Si vous ne voyez pas l'onglet lambda_function.py dans l'éditeur de code, sélectionnez lambda_function.py dans l'explorateur de fichiers, comme le montre le schéma suivant.

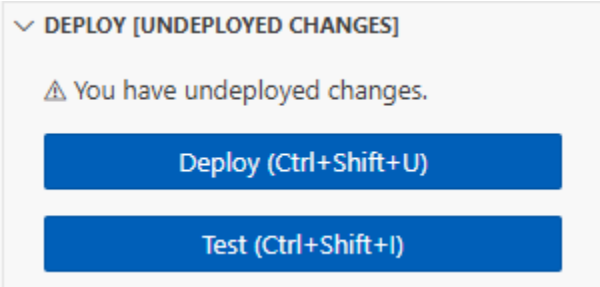


2. Collez le code suivant dans l'onglet lambda_function.py, en remplaçant le code créé par Lambda.

```
import json
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):

    # Get the length and width parameters from the event object. The
    # runtime converts the event object to a Python dictionary
    length = event['length']
    width = event['width']

    area = calculate_area(length, width)
    print(f"The area is {area}")
```

```
    logger.info(f"CloudWatch logs group: {context.log_group_name}")

    # return the calculated area as a JSON string
    data = {"area": area}
    return json.dumps(data)

def calculate_area(length, width):
    return length*width
```

3. Dans la section DÉPLOYER, choisissez Déployer pour mettre à jour le code de votre fonction :



Comprendre le code de votre fonction

Avant de passer à l'étape suivante, examinons le code de la fonction pour comprendre certains concepts Lambda clés.

- Le gestionnaire Lambda :

  Votre fonction Lambda contient une fonction Python nommée `lambda_handler`. Une fonction Lambda en Python peut contenir plusieurs fonctions Python, mais la fonction gestionnaire est toujours le point d'entrée de votre code. Lorsque votre fonction est invoquée, Lambda exécute cette méthode.

  Lorsque vous avez créé votre fonction Hello world à l'aide de la console, Lambda a automatiquement défini le nom de la méthode du gestionnaire pour votre fonction sur `lambda_handler`. Veillez à ne pas modifier le nom de cette fonction Python. Dans ce cas, Lambda ne pourra pas exécuter votre code lorsque vous invoquerez votre fonction.

  Pour en savoir plus sur le gestionnaire Lambda en Python, voir the section called “Handler (Gestionnaire)”.

- L'objet de l'événement Lambda :

La fonction `lambda_handler` accepte deux arguments, `event` et `context`. Un événement dans Lambda est un document au format JSON qui contient des données à traiter par votre fonction.

Si votre fonction est invoquée par une autre personne Service AWS, l'objet d'événement contient des informations sur l'événement à l'origine de l'invocation. Par exemple, si votre fonction est invoquée lorsqu'un objet est chargé dans un compartiment Amazon Simple Storage Service (Amazon S3), l'événement contient le nom du compartiment et la clé de l'objet.

Dans cet exemple, vous allez créer un événement dans la console en saisissant un document au format JSON avec deux paires clé-valeur.

- L'objet de contexte Lambda :

  Le deuxième argument que votre fonction accepte est `context`. Lambda transmet automatiquement l'objet de contexte à votre fonction. L'objet de contexte contient des informations supplémentaires sur l'invocation de la fonction et l'environnement d'exécution.

  Vous pouvez utiliser l'objet de contexte pour générer des informations sur l'invocation de votre fonction à des fins de surveillance. Dans cet exemple, votre fonction utilise le `log_group_name` paramètre pour afficher le nom de son groupe de CloudWatch journaux.

  Pour en savoir plus sur l'objet de contexte Lambda en Python, voir the section called "Contexte".

- Connexion à Lambda :

  Avec Python, vous pouvez utiliser une instruction `print` ou une bibliothèque de journalisation Python pour envoyer des informations au journal de votre fonction. Pour illustrer la différence entre ce qui est capturé, l'exemple de code utilise les deux méthodes. Dans une application de production, nous vous recommandons d'utiliser une bibliothèque de journalisation.

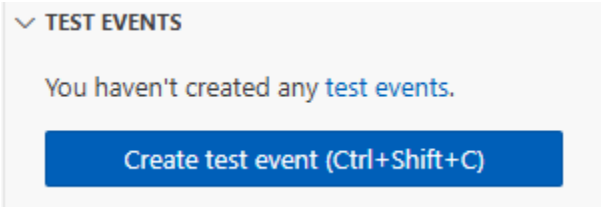
  Pour en savoir plus, veuillez consulter la section the section called "Logging". Pour en savoir plus sur la connexion à d'autres environnements d'exécution, consultez les pages « Génération avec » correspondant aux environnements d'exécution qui vous intéressent.

Invocation de fonction Lambda à l'aide de l'éditeur de code de la console

Pour invoquer votre fonction à l'aide de l'éditeur de code de la console Lambda, créez un événement de test à envoyer à votre fonction. L'événement est un document au format JSON contenant deux paires clé-valeur avec les clés `"length"` et `"width"`.

Pour créer l'événement de test

1. Dans la section ÉVÉNEMENTS DE TEST de l'éditeur de code de console, choisissez Créer un événement de test.



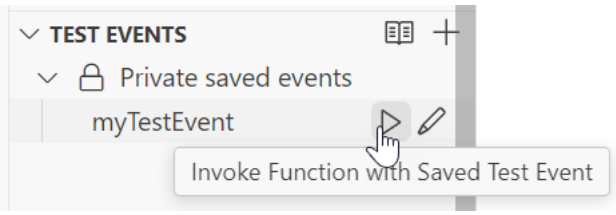
2. Dans Event Name (Nom de l'événement), saisissez **`myTestEvent`**.
3. Dans la section JSON d'événement, remplacez le JSON par défaut par ce qui suit :

```
{
  "length": 6,
  "width": 7
}
```

4. Choisissez Enregistrer.

Pour tester votre fonction et consulter les enregistrements d'invocation

Dans la section ÉVÉNEMENTS DE TEST de l'éditeur de code de la console, cliquez sur l'icône d'exécution à côté de votre événement de test :



Lorsque votre fonction termine son exécution, les journaux des réponses et de la fonction s'affichent dans l'onglet RÉSULTATS. Vous devriez voir des résultats similaires à ce qui suit :

Node.js

```
Status: Succeeded
Test Event Name: myTestEvent

Response
"{\"area\":42}"

Function Logs
START RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a Version: $LATEST
2024-08-31T23:39:45.313Z 5c012b0a-18f7-4805-b2f6-40912935034a INFO The area is 42
2024-08-31T23:39:45.331Z 5c012b0a-18f7-4805-b2f6-40912935034a INFO CloudWatch log
 group:  /aws/lambda/myLambdaFunction
END RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a
REPORT RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a Duration: 20.67 ms Billed
 Duration: 21 ms Memory Size: 128 MB Max Memory Used: 66 MB Init Duration: 163.87 ms

Request ID
5c012b0a-18f7-4805-b2f6-40912935034a
```

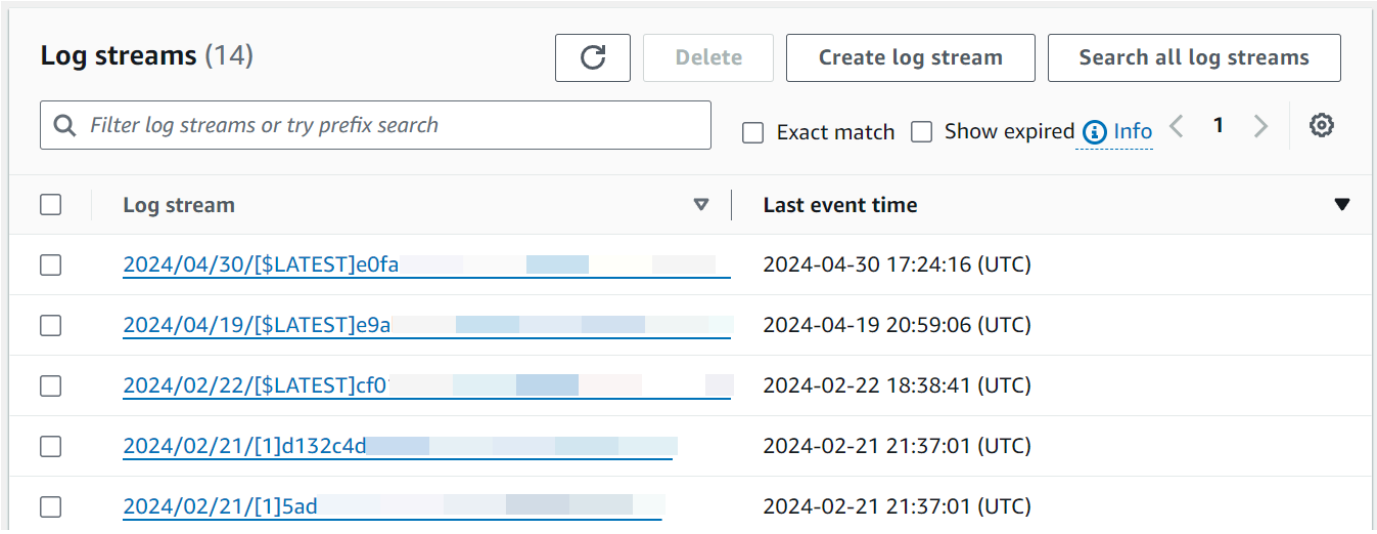
Python

```
Status: Succeeded
Test Event Name: myTestEvent

Response
"{\"area\": 42}"

Function Logs
START RequestId: 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b Version: $LATEST
The area is 42
[INFO] 2024-08-31T23:43:26.428Z 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b CloudWatch logs
 group: /aws/lambda/myLambdaFunction
END RequestId: 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b
REPORT RequestId: 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b Duration: 1.42 ms Billed
 Duration: 2 ms Memory Size: 128 MB Max Memory Used: 39 MB Init Duration: 123.74 ms

Request ID
2d0b1579-46fb-4bf7-a6e1-8e08840eae5b
```

Lorsque vous appelez votre fonction en dehors de la console Lambda, vous devez utiliser CloudWatch Logs pour afficher les résultats d'exécution de votre fonction.

Pour consulter les enregistrements d'invocation de votre fonction dans Logs CloudWatch

1. Ouvrez la page Groupes de journaux de la CloudWatch console.
2. Choisissez le groupe de journaux de votre fonction (`/aws/lambda/myLambdaFunction`). Il s'agit du nom du groupe de journaux que votre fonction a imprimé sur la console.
3. Faites défiler la page vers le bas et choisissez le flux de journaux pour les invocations de fonctions que vous souhaitez consulter.



Vous devez voir des résultats similaires à ce qui suit :

Node.js

```
INIT_START Runtime Version: nodejs:22.v13    Runtime Version ARN:
 arn:aws:lambda:us-
west-2::runtime:e3aaabf6b92ef8755eaae2f4bfdcb7eb8c4536a5e044900570a42bdba7b869d9
START RequestId: aba6c0fc-cf99-49d7-a77d-26d805dacd20 Version: $LATEST
2024-08-23T22:04:15.809Z    5c012b0a-18f7-4805-b2f6-40912935034a  INFO The area
 is 42
2024-08-23T22:04:15.810Z    aba6c0fc-cf99-49d7-a77d-26d805dacd20  INFO
 CloudWatch log group:  /aws/lambda/myLambdaFunction
END RequestId: aba6c0fc-cf99-49d7-a77d-26d805dacd20
REPORT RequestId: aba6c0fc-cf99-49d7-a77d-26d805dacd20    Duration: 17.77 ms
 Billed Duration: 18 ms    Memory Size: 128 MB    Max Memory Used: 67 MB    Init
 Duration: 178.85 ms
```

Python

```
INIT_START Runtime Version: python:3.13.v16    Runtime Version ARN:
 arn:aws:lambda:us-
west-2::runtime:ca202755c87b9ec2b58856efb7374b4f7b655a0ea3deb1d5acc9aee9e297b072
START RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e Version: $LATEST
The area is 42
[INFO] 2024-09-01T00:05:22.464Z 9315ab6b-354a-486e-884a-2fb2972b7d84 CloudWatch
 logs group: /aws/lambda/myLambdaFunction
END RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e
REPORT RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e    Duration: 1.15 ms
 Billed Duration: 2 ms    Memory Size: 128 MB    Max Memory Used: 40 MB
```

Nettoyage

Lorsque vous avez terminé d'utiliser l'exemple de fonction, supprimez-le. Vous pouvez également supprimer le groupe de journaux qui stocke les journaux de la fonction, et le rôle d'exécution créé par la console.

Pour supprimer la fonction Lambda

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction que vous avez créée.
3. Sélectionnez Actions, Supprimer.
4. Saisissez **`confirm`** dans la zone de saisie de texte et choisissez Delete (Supprimer).

Pour supprimer le groupe de journaux

1. Ouvrez la page Log groups (Groupes de journaux) de la console CloudWatch.
2. Sélectionnez le groupe de journaux de la fonction (`/aws/lambda/myLambdaFunction`).
3. Sélectionnez Actions, Delete log group(s) (Supprimer le ou les groupes de journaux).
4. Dans la boîte de dialogue Delete log group(s) (Supprimer le ou les groupes de journaux), sélectionnez Delete (Supprimer).

Pour supprimer le rôle d'exécution

1. Ouvrez la page Rôles de la console Gestion des identités et des accès AWS (IAM).
2. Sélectionnez le rôle d'exécution de la fonction (par exemple, `myLambdaFunction-role-31exxmpl`).
3. Sélectionnez Delete (Supprimer).
4. Dans la fenêtre de dialogue Supprimer le rôle, saisissez le nom du rôle, puis sélectionnez Supprimer.

Ressources supplémentaires et prochaines étapes

Maintenant que vous avez créé et testé une fonction Lambda simple à l'aide de la console, procédez comme suit :

- Découvrez comment ajouter des dépendances à votre fonction et à les déployer à l'aide d'un package de déploiement .zip. Choisissez votre langage préféré à partir des liens suivants.

  Node.js

  the section called "Déploiement d'archives de fichiers .zip"

  Typescript

  the section called "Déployez des archives de fichiers .zip"

  Python

  the section called "Déploiement d'archives de fichiers .zip"

  Ruby

  the section called "Déploiement d'archives de fichiers .zip"

  Java

  the section called "Déploiement d'archives de fichiers .zip"

  Go

  the section called "Déploiement d'archives de fichiers .zip"

  C#

  the section called "Package de déploiement"

- Pour savoir comment appeler une fonction Lambda à l'aide d'une autre Service AWS, consultez. Didacticiel : utilisation d’un déclencheur Amazon S3 pour invoquer une fonction Lambda
- Choisissez l’un des tutoriels suivants pour un exemple plus complexe d’utilisation de Lambda avec d’autres Services AWS.
  - Didacticiel : Utiliser Lambda avec Amazon API Gateway : créer une API REST Amazon API Gateway qui invoque une fonction Lambda.
  - Utilisation d'une fonction Lambda pour accéder à une base de données Amazon RDS: utilisez une fonction Lambda pour écrire des données dans une base de données Amazon Relational Database Service (Amazon RDS) via un proxy RDS.
  - Utilisation d'un déclencheur Amazon S3 pour créer des images miniatures : utilisez une fonction Lambda pour créer une miniature chaque fois qu'un fichier image est chargé dans un compartiment Amazon S3.

Commencer avec des exemples d’applications et de modèles

Les ressources suivantes peuvent être utilisées pour créer et déployer rapidement des applications sans serveur qui mettent en œuvre certains cas d’utilisation courants de Lambda. Pour chacun des exemples d'applications, nous fournissons des instructions pour créer et configurer des ressources manuellement à l'aide de AWS Management Console, ou pour déployer les ressources AWS Serverless Application Model à l'aide d'iAc. Suivez les instructions de la console pour en savoir plus sur la configuration des AWS ressources individuelles pour chaque application, ou utilisez-les pour AWS SAM déployer rapidement des ressources comme vous le feriez dans un environnement de production.

Traitement des fichiers

- Application de chiffrement de PDF : créez une application sans serveur qui chiffre les fichiers PDF lorsqu’ils sont chargés dans un compartiment Amazon Simple Storage Service et les enregistre dans un autre compartiment, ce qui est utile pour sécuriser les documents sensibles lors de leur chargement.

- Application d’analyse d’images : créez une application sans serveur qui extrait le texte des images à l’aide d’Amazon Rekognition, ce qui est utile pour le traitement des documents, la modération de contenu et l’analyse automatique des images.

Intégration de base de données

- Queue-to-Database Application : créez une application sans serveur qui écrit des messages de file d'attente dans une base de données Amazon RDS, ce qui est utile pour traiter les inscriptions des utilisateurs et gérer les soumissions de commandes.

- Gestionnaire d’événements de base de données : créez une application sans serveur qui répond aux modifications des tables Amazon DynamoDB, ce qui est utile pour la journalisation des audits, la réplication des données et les flux de travail automatisés.

Tâches planifiées

- Application de maintenance de base de données : créez une application sans serveur qui supprime automatiquement les entrées datant de plus de 12 mois d'une table Amazon DynamoDB à l'aide d'une planification cron, ce qui est utile pour la maintenance automatisée des bases de données et la gestion du cycle de vie des données.
- Créez une règle EventBridge planifiée pour les fonctions Lambda : utilisez des expressions planifiées pour les règles EventBridge afin de déclencher une fonction Lambda selon un calendrier chronométré. Ce format utilise la syntaxe cron et peut être défini avec une granularité d'une minute.

Des flux de travail de longue durée

- Application de traitement des commandes : créez une application sans serveur à l'aide de fonctions durables qui gère le traitement des commandes complexes, notamment le traitement des paiements, les vérifications d'inventaire et la coordination des expéditions. Cet exemple montre comment créer des flux de travail qui peuvent s'exécuter pendant de longues périodes tout en conservant leur état.

Ressources supplémentaires

Utilisez les ressources suivantes pour explorer plus en détail le développement d'applications Lambda et sans serveur :

- Serverless Land : une bibliothèque de ready-to-use modèles pour créer des applications sans serveur. Il aide les développeurs à créer des applications plus rapidement à l'aide de AWS services tels que Lambda, API Gateway et. EventBridge Le site propose des solutions prédéfinies et des bonnes pratiques, afin de faciliter le développement de systèmes sans serveur.
- Exemples d'applications Lambda : applications disponibles dans le GitHub référentiel de ce guide. Ces exemples montrent l'utilisation de différentes langues et de différents AWS services. Chaque exemple d'application comprend des scripts facilitant le déploiement et le nettoyage, ainsi que des ressources complémentaires.
- Exemples de code pour Lambda à l'aide de AWS SDKs : exemples qui vous montrent comment utiliser Lambda avec des kits de développement AWS logiciel (). SDKs Ces exemples incluent les éléments de base, les actions, les scénarios et les contributions de AWS la communauté. Les

exemples couvrent les opérations essentielles, les fonctions de service individuelles et les tâches spécifiques utilisant plusieurs fonctions ou AWS services.

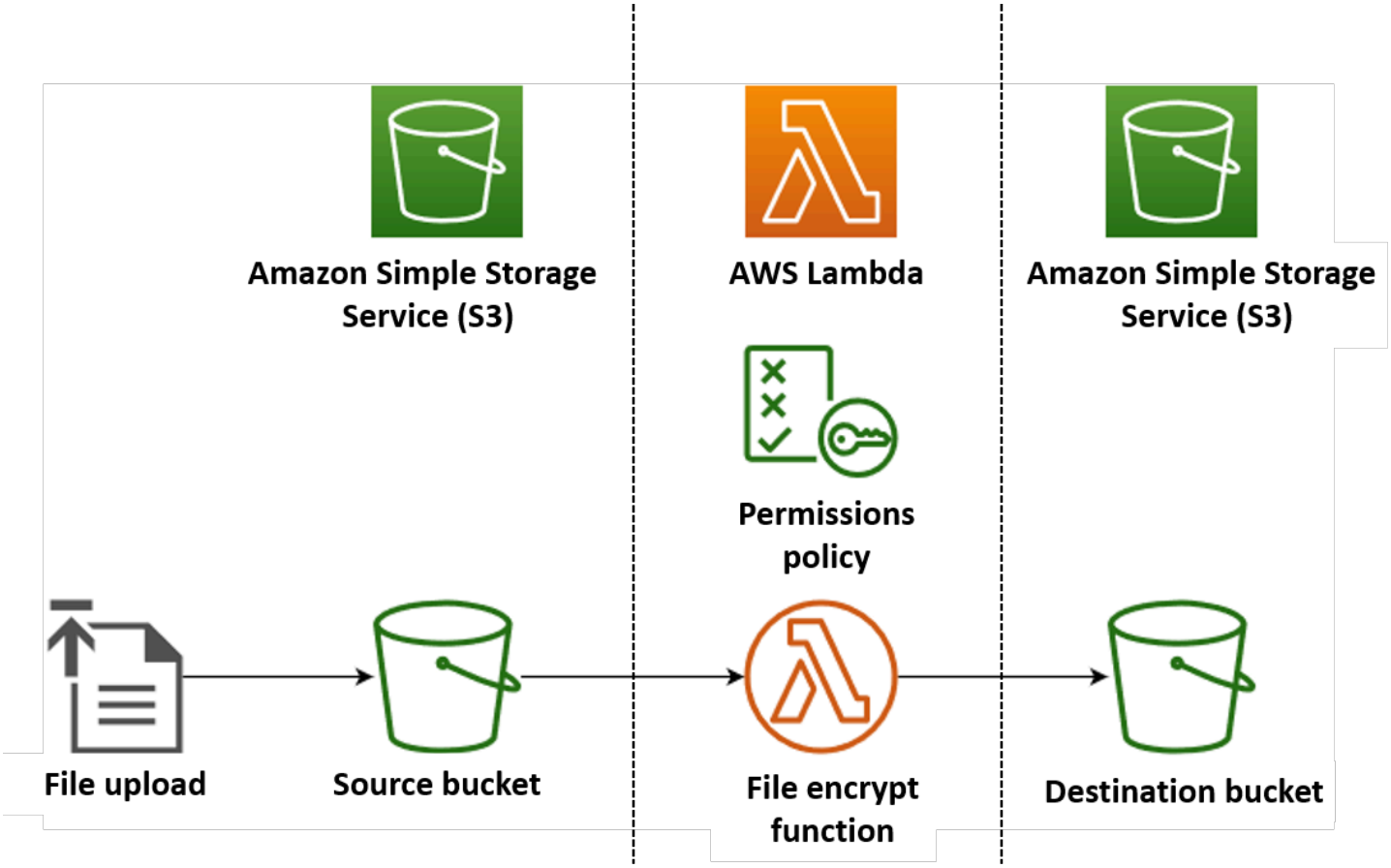
Création d’une application sans serveur de traitement de fichiers

L’un des cas d’utilisation les plus courants de Lambda consiste à effectuer des tâches de traitement de fichiers. Par exemple, vous pouvez utiliser une fonction Lambda pour créer automatiquement des fichiers PDF à partir de fichiers HTML ou d’images, ou pour créer des miniatures lorsqu’un utilisateur télécharge une image.

Dans cet exemple, vous créez une application qui chiffre automatiquement les fichiers PDF lorsqu’ils sont chargés dans un compartiment Amazon Simple Storage Service (Amazon S3). Pour mettre en œuvre cette application, vous créez les ressources suivantes :

- Un compartiment S3 dans lequel les utilisateurs peuvent charger des fichiers PDF
- Une fonction Lambda en Python qui lit le fichier téléchargé et en crée une version cryptée et protégée par mot de passe
- Un deuxième compartiment S3 dans lequel Lambda pourra enregistrer le fichier chiffré

Vous créez également une politique Gestion des identités et des accès AWS (IAM) pour autoriser votre fonction Lambda à effectuer des opérations de lecture et d'écriture sur vos compartiments S3.



> **Tip**
>
> Si vous utilisez Lambda pour la première fois, nous vous recommandons de commencer avec le didacticiel Création de votre première fonction avant de créer cet exemple d'application.

Vous pouvez déployer votre application manuellement en créant et en configurant des ressources à l'aide du AWS Management Console ou du AWS Command Line Interface (AWS CLI). Vous pouvez également déployer l'application en utilisant le AWS Serverless Application Model (AWS SAM). AWS SAM est un outil d'infrastructure en tant que code (IaC). Avec l'IaC, vous ne créez pas de ressources manuellement, mais vous les définissez dans le code, puis vous les déployez automatiquement.

Si vous souhaitez en savoir plus sur l'utilisation de Lambda avec l'IaC avant de déployer cet exemple d'application, consultez the section called "Infrastructure en tant que code (IaC)".

Créer les fichiers de code source de la fonction Lambda

Créez les fichiers suivants dans le répertoire de votre projet :

- `lambda_function.py` : le code de fonction Python de la fonction Lambda qui effectue le chiffrement de fichiers
- `requirements.txt` : un fichier manifeste définissant les dépendances requises par le code de votre fonction Python

Développez les sections suivantes pour afficher le code et pour en savoir plus sur le rôle de chaque fichier. Pour créer les fichiers sur votre machine locale, copiez et collez le code ci-dessous ou téléchargez les fichiers depuis le aws-lambda-developer-guide GitHub dépôt.

Code de fonction Python

Copiez et collez le code suivant dans un fichier nommé `lambda_function.py`.

```
from pypdf import PdfReader, PdfWriter
import uuid
import os
from urllib.parse import unquote_plus
import boto3

# Create the S3 client to download and upload objects from S3
s3_client = boto3.client('s3')

def lambda_handler(event, context):
    # Iterate over the S3 event object and get the key for all uploaded files
    for record in event['Records']:
        bucket = record['s3']['bucket']['name']
        key = unquote_plus(record['s3']['object']['key']) # Decode the S3 object key to
 remove any URL-encoded characters
        download_path = f'/tmp/{uuid.uuid4()}.pdf' # Create a path in the Lambda tmp
 directory to save the file to
        upload_path = f'/tmp/converted-{uuid.uuid4()}.pdf' # Create another path to
 save the encrypted file to

        # If the file is a PDF, encrypt it and upload it to the destination S3 bucket
        if key.lower().endswith('.pdf'):
            s3_client.download_file(bucket, key, download_path)
            encrypt_pdf(download_path, upload_path)
            encrypted_key = add_encrypted_suffix(key)
            s3_client.upload_file(upload_path, f'{bucket}-encrypted', encrypted_key)

# Define the function to encrypt the PDF file with a password
def encrypt_pdf(file_path, encrypted_file_path):
```

```
    reader = PdfReader(file_path)
    writer = PdfWriter()

    for page in reader.pages:
        writer.add_page(page)

    # Add a password to the new PDF
    writer.encrypt("my-secret-password")

    # Save the new PDF to a file
    with open(encrypted_file_path, "wb") as file:
        writer.write(file)

# Define a function to add a suffix to the original filename after encryption
def add_encrypted_suffix(original_key):
    filename, extension = original_key.rsplit('.', 1)
    return f'{filename}_encrypted.{extension}'
```

> **Note**
>
> Dans cet exemple de code, un mot de passe pour le fichier chiffré (`my-secret-password`) est codé en dur dans le code de fonction. Dans une application de production, n'incluez pas d'informations sensibles telles que des mots de passe dans votre code de fonction. Créez plutôt un AWS Secrets Manager secret, puis utilisez l'extension Lambda AWS Parameters and Secrets pour récupérer vos informations d'identification dans votre fonction Lambda.

Le code de la fonction python contient trois fonctions : la fonction de gestion que Lambda exécute lorsque votre fonction est invoquée, et deux fonctions distinctes nommées `add_encrypted_suffix` et `encrypt_pdf` que le gestionnaire appelle pour effectuer le chiffrement du PDF.

Lorsque votre fonction est invoquée par Amazon S3, Lambda transmet un argument d'événement au format JSON à la fonction contenant des informations sur l'événement à l'origine de l'invocation. Dans ce cas, les informations incluent le nom du compartiment S3 et les clés d'objet pour les fichiers téléchargés. Pour de plus amples informations sur le format de l'objet d'événement pour Amazon S3, consultez the section called "S3".

Votre fonction utilise ensuite le AWS SDK pour Python (Boto3) pour télécharger les fichiers PDF spécifiés dans l'objet d'événement vers son répertoire de stockage temporaire local, avant de les chiffrer à l'aide de la `pypdf`bibliothèque.

Enfin, la fonction utilise le kit SDK Boto3 pour stocker le fichier chiffré dans votre compartiment de destination S3.

Fichier manifeste **`requirements.txt`**

Copiez et collez le code suivant dans un fichier nommé `requirements.txt`.

```
boto3
pypdf
```

Dans cet exemple, le code de votre fonction ne comporte que deux dépendances qui ne font pas partie de la bibliothèque Python standard : le kit SDK pour Python (Boto3) et le package `pypdf` utilisé par la fonction pour effectuer le chiffrement du PDF.

> **Note**
>
> Une version du kit SDK pour Python (Boto3) est incluse dans l'environnement d'exécution Lambda, de sorte que votre code s'exécute sans ajouter Boto3 au package de déploiement de votre fonction. Toutefois, pour garder le contrôle total des dépendances de votre fonction et éviter d'éventuels problèmes de désalignement de version, la bonne pratique pour Python consiste à inclure toutes les dépendances de fonction dans le package de déploiement de votre fonction. Pour en savoir plus, consultez the section called "Dépendances d'exécution dans Python".

Déployer l'application

Vous pouvez créer et déployer les ressources pour cet exemple d'application manuellement ou à l'aide de AWS SAM. Dans un environnement de production, nous vous recommandons d'utiliser un outil IaC AWS SAM pour déployer rapidement et de manière répétitive des applications complètes sans serveur sans recourir à des processus manuels.

Déploiement manuel des ressources

Pour déployer votre application manuellement :

- Créez les compartiments Amazon S3 source et destination
- Créez une fonction Lambda qui chiffre un fichier PDF et enregistre la version chiffrée dans un compartiment S3

- Configurez un déclencheur Lambda qui invoque votre fonction lorsque des objets sont chargés dans votre compartiment source

Avant de commencer, assurez-vous que Python est installé sur votre machine de génération.

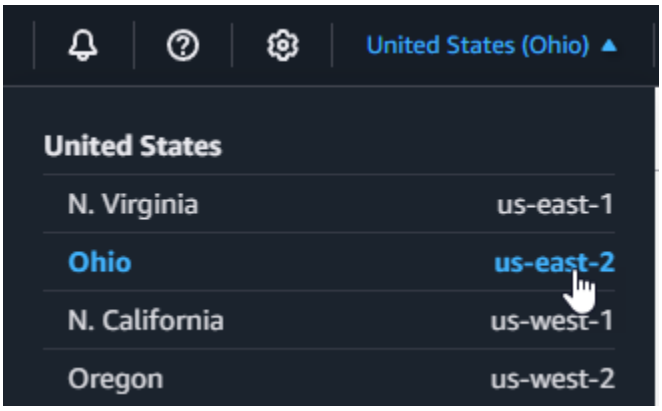
Création de deux compartiments S3

Créez deux compartiments S3. Le premier compartiment est le compartiment source dans lequel vous allez charger vos fichiers PDF. Le second compartiment est utilisé par Lambda pour enregistrer les fichiers chiffrés lorsque vous invoquez votre fonction.

Console

Pour créer les compartiments S3 (console)

1. Ouvrez la page Compartiments à usage général de la console Amazon S3.
2. Sélectionnez le Région AWS plus proche de votre situation géographique. Vous pouvez modifier votre région à l'aide de la liste déroulante en haut de l'écran.



3. Choisissez Create bucket (Créer un compartiment).
4. Sous Configuration générale, procédez comme suit :
   a. Pour le type de compartiment, veillez à sélectionner Usage général.
   b. Pour le nom du compartiment, saisissez un nom unique au monde qui respecte les règles de dénomination du compartiment Amazon S3. Les noms de compartiment peuvent contenir uniquement des lettres minuscules, des chiffres, de points (.) et des traits d'union (-).
5. Conservez les valeurs par défaut de toutes les autres options et choisissez Créer un compartiment.

6. Répétez les étapes 1 à 4 pour créer votre compartiment de destination. Pour Nom du compartiment, saisissez **amzn-s3-demo-bucket-encrypted**, où **amzn-s3-demo-bucket** est le nom du compartiment source que vous venez de créer.

AWS CLI

Avant de commencer, assurez-vous d'avoir installéAWS CLI sur la machine de génération.

Pour créer les compartiments Amazon S3 (AWS CLI)

1. Exécutez la commande CLI suivante pour créer votre compartiment source. Le nom que vous choisissez pour votre compartiment doit être unique au monde et respecter les Règles de dénomination du compartiment Amazon S3. Les noms peuvent contenir uniquement des lettres minuscules, des chiffres, de points (.) et des traits d'union (-). Pour `region` et `LocationConstraint`, choisissez la Région AWS la plus proche de votre emplacement géographique.

```
aws s3api create-bucket --bucket amzn-s3-demo-bucket --region us-east-2 \
--create-bucket-configuration LocationConstraint=us-east-2
```

   Plus loin dans le didacticiel, vous devez créer votre fonction Lambda dans le même Région AWS emplacement que votre compartiment source. Notez donc la région que vous avez choisie.

2. Exécutez la commande suivante pour créer votre compartiment de destination. Pour le nom du compartiment, vous devez utiliser **amzn-s3-demo-bucket-encrypted**, où **amzn-s3-demo-bucket** est le nom du compartiment source que vous avez créé à l'étape 1. Pour `region` et`LocationConstraint`, choisissez le même Région AWS que celui que vous avez utilisé pour créer votre bucket source.

```
aws s3api create-bucket --bucket amzn-s3-demo-bucket-encrypted --region us-
east-2 \
--create-bucket-configuration LocationConstraint=us-east-2
```

Créer un rôle d'exécution

Un rôle d'exécution est un rôle IAM qui accorde à une fonction Lambda l'autorisation d' Services AWS accès et de ressources. Pour donner à votre fonction un accès en lecture et en écriture à Amazon S3, vous attachez la politique gérée par AWS `AmazonS3FullAccess`.

Console

Créer un rôle IAM et l'attacher à la politique gérée par **`AmazonS3FullAccess`** (console)

1. Ouvrez la page Roles (Rôles) dans la console IAM.
2. Choisissez Créer un rôle.
3. Pour Type d'entité approuvée, sélectionnez Service AWS , et pour Cas d'utilisation, sélectionnez Lambda.
4. Choisissez Suivant.
5. Ajoutez la politique gérée `AmazonS3FullAccess` en procédant comme suit :
   a. Dans Politiques d'autorisations, saisissez **`AmazonS3FullAccess`** dans la barre de recherche.
   b. Activez la case à cocher en regard de la stratégie.
   c. Choisissez Suivant.
6. Dans Détails du rôle, saisissez **`LambdaS3Role`** pour Nom du rôle.
7. Choisissez Create Role (Créer un rôle).

AWS CLI

Pour créer un rôle IAM et l'attacher à la politique gérée par **`AmazonS3FullAccess`** (AWS CLI)

1. Enregistrez le JSON suivant dans un fichier nommé `trust-policy.json`. Cette politique de confiance permet à Lambda d'utiliser les autorisations du rôle en autorisant le principal `lambda.amazonaws.com` du service à appeler l'action AWS Security Token Service (AWS STS)`AssumeRole`.

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
```

```
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

2. À partir du répertoire dans lequel vous avez enregistré le document de politique d'approbation JSON, exécutez la commande CLI suivante pour créer le rôle d'exécution.

```
aws iam create-role --role-name LambdaS3Role --assume-role-policy-document
 file://trust-policy.json
```

3. Pour associer la politique gérée `AmazonS3FullAccess`, exécutez la commande de la CLI suivante.

```
aws iam attach-role-policy --role-name LambdaS3Role --policy-arn
 arn:aws:iam::aws:policy/AmazonS3FullAccess
```

Créer le package de déploiement de la fonction

Pour créer votre fonction, vous créez un package de déploiement contenant le code de votre fonction et ses dépendances. Pour cette application, votre code de fonction utilise une bibliothèque distincte pour le chiffrement du PDF.

Pour créer le package de déploiement

1. Accédez au répertoire du projet contenant les `requirements.txt` fichiers `lambda_function.py` et que vous avez créés ou téléchargés GitHub précédemment et créez un nouveau répertoire nommé`package`.
2. Installez les dépendances spécifiées dans le fichier `requirements.txt` de votre répertoire `package` en exécutant la commande suivante.

```
pip install -r requirements.txt --target ./package/
```

3. Créez un fichier .zip contenant le code de votre application et ses dépendances. Sous Linux ou macOS, exécutez les commandes suivantes depuis votre interface de ligne de commande.

```
cd package
zip -r ../lambda_function.zip .
cd ..
zip lambda_function.zip lambda_function.py
```

Sous Windows, utilisez l'outil zip de votre choix pour créer le fichier `lambda_function.zip`. Assurez-vous que votre fichier `lambda_function.py` et les dossiers contenant vos dépendances sont installés à la racine du fichier .zip.

Vous pouvez également créer votre package de déploiement à l'aide d'un environnement virtuel Python. Consultez Travailler avec des archives de fichiers .zip pour les fonctions Lambda Python

Créer la fonction Lambda

Vous utilisez maintenant le package de déploiement que vous avez créé à l'étape précédente pour déployer votre fonction Lambda.

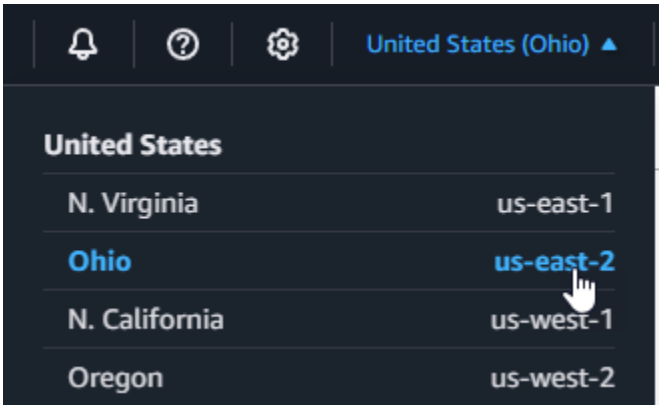
Console

Pour créer une fonction (console)

Pour créer votre fonction Lambda à l'aide de la console, vous devez d'abord créer une fonction de base contenant du code « Hello world ». Vous remplacez ensuite ce code par votre propre code de fonction en chargeant le fichier .zip que vous avez créé à l'étape précédente.

Pour garantir que votre fonction n'expire pas lors du chiffrement de fichiers PDF volumineux, configurez les paramètres de mémoire et de délai d'expiration de la fonction. Définissez également le format de journal de la fonction sur JSON. La configuration de journaux au format JSON est nécessaire lors de l'utilisation du script de test fourni afin que celui-ci puisse lire l'état d'invocation de la fonction dans CloudWatch Logs pour confirmer la réussite de l'appel.

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Assurez-vous de travailler dans le même environnement que celui dans Région AWS lequel vous avez créé votre compartiment S3. Vous pouvez modifier votre région à l'aide de la liste déroulante en haut de l'écran.



3. Choisissez Créer une fonction.
4. Choisissez Créer à partir de zéro.
5. Sous Informations de base, procédez comme suit :
   a. Sous Nom de la fonction, saisissez **`EncryptPDF`**.
   b. Pour Environnement d'exécution, choisissez Python 3.12.
   c. Pour Architecture, choisissez x86_64.
6. Associez le rôle d'exécution que vous avez créé à l'étape précédente en procédant comme suit :
   a. Développez la section Changer le rôle d'exécution par défaut.
   b. Sélectionnez Utiliser un rôle existant.
   c. Sous Rôle existant, sélectionnez votre rôle (`LambdaS3Role`).
7. Choisissez Créer une fonction.

Pour charger le code de fonction (console)

1. Dans le volet Source du code, choisissez Charger à partir de.
2. Choisissez Fichier .zip.
3. Choisissez Charger.
4. Dans le sélecteur de fichiers, sélectionnez votre fichier .zip et choisissez Ouvrir.
5. Choisissez Enregistrer.

Pour configurer la mémoire et le délai d'expiration d'une fonction (console)

1. Sélectionnez l'onglet Configuration correspondant à votre fonction.

2. Dans le volet de Configuration générale, choisissez Modifier.
3. Réglez Mémoire sur 256 Mo et Délai d'expiration sur 15 secondes.
4. Choisissez Enregistrer.

Pour configurer le format de journal (console)

1. Sélectionnez l'onglet Configuration correspondant à votre fonction.
2. Sélectionnez Outils de surveillance et d'exploitation.
3. Dans le volet de configuration de la journalisation, choisissez Modifier.
4. Pour Configuration de journalisation, sélectionnez JSON.
5. Choisissez Enregistrer.

AWS CLI

Pour créer la fonction (AWS CLI)

- Exécutez la commande suivante depuis le répertoire contenant votre fichier `lambda_function.zip`. Pour le paramètre `region`, remplacez `us-east-2` par la région dans laquelle vous avez créé vos compartiments S3.

```
aws lambda create-function --function-name EncryptPDF \
--zip-file fileb://lambda_function.zip --handler lambda_function.lambda_handler
 \
--runtime python3.12 --timeout 15 --memory-size 256 \
--role arn:aws:iam::123456789012:role/LambdaS3Role --region us-east-2 \
--logging-config LogFormat=JSON
```

Configuration d'un déclencheur Amazon S3 pour invoquer la fonction

Pour que votre fonction Lambda s'exécute lorsque vous chargez un fichier dans votre compartiment source, vous devez configurer un déclencheur pour votre fonction. Vous pouvez configurer le déclencheur Amazon S3 à l'aide de la console ou de la AWS CLI.

> **Important**
>
> Cette procédure configure le compartiment S3 pour qu'il invoque votre fonction chaque fois qu'un objet est créé dans le compartiment. Veillez à configurer cela uniquement pour le

compartiment source. Si votre fonction Lambda crée des objets dans le même compartiment que celui qui l'invoque, votre fonction peut être invoquée en continu dans une boucle. Cela peut entraîner la facturation de frais imprévus à votre Compte AWS charge.

Console

Pour configurer le déclencheur Amazon S3 (console)

1. Ouvrez la page Fonctions de la console Lambda et choisissez votre fonction (`EncryptPDF`).
2. Choisissez Add trigger (Ajouter déclencheur).
3. Sélectionnez S3.
4. Sous Compartiment, sélectionnez votre compartiment source.
5. Sous Types d'événements, sélectionnez Tous les événements de création d'objets.
6. Sous Invocation récursive, cochez la case pour confirmer qu'il n'est pas recommandé d'utiliser le même compartiment S3 pour les entrées et les sorties. Vous pouvez en savoir plus sur les modèles d'invocation récursive dans Lambda en lisant la rubrique Modèles récursifs qui provoquent des fonctions Lambda incontrôlables dans Serverless Land.
7. Choisissez Ajouter.

   Lorsque vous créez un déclencheur à l'aide de la console Lambda, ce dernier crée automatiquement une politique basée sur une ressource pour donner au service que vous sélectionnez l'autorisation d'invoquer votre fonction.

AWS CLI

Pour configurer le déclencheur Amazon S3 (AWS CLI)

1. Ajoutez une politique basée sur les ressources à votre fonction qui permet à votre compartiment source Amazon S3 d'invoquer votre fonction lorsque vous ajoutez un fichier. Une déclaration de politique basée sur les ressources donne à d'autres personnes Services AWS l'autorisation d'invoquer votre fonction. Pour autoriser Amazon S3 à invoquer votre fonction, exécutez la commande CLI suivante. Assurez-vous de remplacer le `source-account` paramètre par votre propre Compte AWS identifiant et d'utiliser votre propre nom de compartiment source.

   ```
   aws lambda add-permission --function-name EncryptPDF \
   ```

```
--principal s3.amazonaws.com --statement-id s3invoke --action
 "lambda:InvokeFunction" \
--source-arn arn:aws:s3:::amzn-s3-demo-bucket \
--source-account 123456789012
```

La politique que vous définissez à l'aide de cette commande permet à Amazon S3 d'invoquer votre fonction uniquement lorsqu'une action a lieu sur votre compartiment source.

> **Note**
>
> Bien que les noms des compartiments S3 soient globalement uniques, il est préférable de spécifier que le compartiment doit appartenir à votre compte lorsque vous utilisez des politiques basées sur les ressources. En effet, si vous supprimez un bucket, il est possible qu'un autre Compte AWS en crée un avec le même Amazon Resource Name (ARN).

2. Enregistrez le JSON suivant dans un fichier nommé `notification.json`. Lorsqu'il est appliqué à votre compartiment source, ce JSON configure le compartiment pour qu'il envoie une notification à votre fonction Lambda chaque fois qu'un nouvel objet est ajouté. Remplacez le Compte AWS numéro et, Région AWS dans la fonction Lambda, l'ARN par votre propre numéro de compte et votre propre région.

```
{
"LambdaFunctionConfigurations": [
    {
      "Id": "EncryptPDFEventConfiguration",
      "LambdaFunctionArn": "arn:aws:lambda:us-
east-2:123456789012:function:EncryptPDF",
      "Events": [ "s3:ObjectCreated:Put" ]
    }
  ]
}
```

3. Exécutez la commande CLI suivante pour appliquer les paramètres de notification du fichier JSON que vous avez créé à votre compartiment source. Remplacez `amzn-s3-demo-bucket` par le nom de votre propre compartiment source.

```
aws s3api put-bucket-notification-configuration --bucket amzn-s3-demo-bucket \
--notification-configuration file://notification.json
```

Pour en savoir plus sur la `put-bucket-notification-configuration` commande et l'`notification-configuration`option, consultez le manuel de référence put-bucket-notification-configurationdes commandes de la AWS CLI.

Déployez les ressources à l'aide de AWS SAM

Avant de commencer, assurez-vous que Docker et la dernière version de AWS SAMCLI sont installés sur votre machine de génération.

1. Dans le répertoire de votre projet, copiez et collez le code suivant dans un fichier nommé `template.yaml`. Remplacez les espaces réservés des noms des compartiments :

   - Pour le compartiment source, remplacez `amzn-s3-demo-bucket` par un nom conforme aux règles de dénomination des compartiments S3.
   - Pour le compartiment de destination, remplacez `amzn-s3-demo-bucket-encrypted` par `<source-bucket-name>-encrypted`, où `<source-bucket>` est le nom que vous avez choisi pour votre compartiment source.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31

Resources:
  EncryptPDFFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: EncryptPDF
      Architectures: [x86_64]
      CodeUri: ./
      Handler: lambda_function.lambda_handler
      Runtime: python3.12
      Timeout: 15
      MemorySize: 256
      LoggingConfig:
        LogFormat: JSON
      Policies:
        - AmazonS3FullAccess
      Events:
        S3Event:
          Type: S3
```

```
          Properties:
            Bucket: !Ref PDFSourceBucket
            Events: s3:ObjectCreated:*

  PDFSourceBucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: amzn-s3-demo-bucket

  EncryptedPDFBucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: amzn-s3-demo-bucket-encrypted
```

Le AWS SAM modèle définit les ressources que vous créez pour votre application. Dans cet exemple, le modèle définit une fonction Lambda utilisant le type `AWS::Serverless::Function` et deux compartiments S3 utilisant le type `AWS::S3::Bucket`. Les noms de compartiments spécifiés dans le modèle sont des espaces réservés. Avant de déployer l'application à l'aide de AWS SAM, vous devez modifier le modèle pour renommer les compartiments avec des noms uniques au niveau mondial conformes aux règles de dénomination des compartiments S3. Cette étape est expliquée plus en détail dans the section called “Déployez les ressources à l'aide de AWS SAM”.

La définition de la ressource de fonction Lambda configure un déclencheur pour la fonction à l’aide de la propriété d’événement `S3Event`. Ce déclencheur entraîne l’invocation de votre fonction chaque fois qu’un objet est créé dans votre compartiment source.

La définition de la fonction spécifie également une politique Gestion des identités et des accès AWS (IAM) à associer au rôle d'exécution de la fonction. La politique gérée par AWS `AmazonS3FullAccess` donne à votre fonction les autorisations dont elle a besoin pour lire et écrire des objets sur Amazon S3.

2. Exécutez la commande suivante dans le répertoire où vous avez enregistré les fichiers `template.yaml`, `lambda_function.py` et `requirements.txt`.

```
sam build --use-container
```

Cette commande rassemble les artefacts de compilation pour votre application et les place dans le format et l’emplacement appropriés pour les déployer. En spécifiant l’option `--use-container`, vous créez votre fonction à l’intérieur d’un conteneur Docker de type Lambda. Nous

l'utilisons ici, vous n'avez donc pas besoin d'installer Python 3.12 sur votre machine locale pour que la compilation fonctionne.

Pendant le processus de génération, AWS SAM recherche le code de la fonction Lambda à l'emplacement que vous avez spécifié avec la `CodeUri` propriété dans le modèle. Dans ce cas, nous avons spécifié le répertoire actuel comme emplacement (`./`).

Si un `requirements.txt` fichier est présent, AWS SAM utilisez-le pour rassembler les dépendances spécifiées. Par défaut, AWS SAM crée un package de déploiement .zip avec le code de votre fonction et ses dépendances. Vous pouvez également choisir de déployer votre fonction sous forme d'image de conteneur à l'aide de la PackageTypepropriété.

3. Pour déployer votre application et créer les ressources Lambda et Amazon S3 spécifiées dans votre AWS SAM modèle, exécutez la commande suivante.

```
sam deploy --guided
```

L'utilisation du `--guided` drapeau signifie que des instructions vous AWS SAM seront affichées pour vous guider tout au long du processus de déploiement. Pour ce déploiement, acceptez les options par défaut en appuyant sur Entrée.

Au cours du processus de déploiement, AWS SAM crée les ressources suivantes dans votre Compte AWS :

- Une CloudFormation pile nommée `sam-app`
- Une fonction Lambda nommée `EncryptPDF`
- Deux compartiments S3 portant les noms que vous avez choisis lorsque vous avez modifié le `template.yaml` AWS SAM fichier modèle
- Un rôle d'exécution IAM pour votre fonction avec le format de nom `sam-app-EncryptPDFFunctionRole-2qGaapHFWOQ8`

Lorsque vous avez AWS SAM terminé de créer vos ressources, le message suivant devrait s'afficher :

```
Successfully created/updated stack - sam-app in us-east-2
```

Tester l'application

Pour tester votre application, chargez un fichier PDF dans votre compartiment source et vous confirmez que Lambda crée une version chiffrée du fichier dans votre compartiment de destination. Dans cet exemple, vous pouvez le tester manuellement à l'aide de la AWS CLI console ou du script de test fourni.

Pour les applications de production, vous pouvez utiliser des méthodes et techniques de test traditionnelles, telles que les tests unitaires, pour confirmer le bon fonctionnement de votre code de fonction Lambda. Les pratiques exemplaires consistent également à effectuer des tests comme ceux du script de test fourni, qui réalisent des tests d'intégration avec des ressources réelles, basées sur le cloud. Les tests d'intégration dans le cloud confirment que votre infrastructure a été correctement déployée et que les événements circulent entre les différents services comme prévu. Pour en savoir plus, veuillez consulter la section Test de fonctions sans serveur.

Test manuel de l'application

Vous pouvez tester votre fonction manuellement en ajoutant un fichier PDF à votre compartiment source Amazon S3. Lorsque vous ajoutez votre fichier au compartiment source, votre fonction Lambda doit être automatiquement invoquée et doit stocker une version chiffrée du fichier dans votre compartiment cible.

Console

Pour tester votre application en chargeant un fichier (console)

1. Pour charger un fichier PDF dans votre compartiment S3, procédez comme suit :
   a. Ouvrez la page Compartiments de la console Amazon S3 et choisissez votre compartiment source.
   b. Choisissez Charger.
   c. Choisissez Ajouter des fichiers et utilisez le sélecteur de fichiers pour sélectionner le fichier PDF que vous souhaitez charger.
   d. Choisissez Ouvrir, puis Charger.
2. Vérifiez que Lambda a enregistré une version chiffrée de votre fichier PDF dans votre compartiment cible en procédant comme suit :
   a. Retournez à la page Compartiments de la console Amazon S3 et choisissez votre compartiment de destination.

b. Dans le volet Objets, vous devriez maintenant voir un fichier au format de nom `filename_encrypted.pdf` (où `filename.pdf` était le nom du fichier que vous avez chargé dans votre compartiment source). Pour télécharger votre PDF chiffré, sélectionnez le fichier, puis choisissez Télécharger.

c. Vérifiez que vous pouvez ouvrir le fichier téléchargé avec le mot de passe avec lequel votre fonction Lambda l'a protégé (`my-secret-password`).

AWS CLI

Pour tester votre application en chargeant un fichier (AWS CLI)

1. À partir du répertoire contenant le fichier PDF que vous souhaitez charger, exécutez la commande de la CLI suivante. Remplacez le paramètre `--bucket` par le nom de votre compartiment source. Pour les paramètres `--key` et `--body`, utilisez le nom de fichier de votre fichier de test.

```
aws s3api put-object --bucket amzn-s3-demo-bucket --key test.pdf --body ./
test.pdf
```

2. Vérifiez que votre fonction a créé une version chiffrée de votre fichier et l'a enregistrée dans votre compartiment S3 cible. Exécutez la commande CLI suivante, en remplaçant `amzn-s3-demo-bucket-encrypted` par le nom de votre compartiment de destination.

```
aws s3api list-objects-v2 --bucket amzn-s3-demo-bucket-encrypted
```

Si votre fonction s'exécute correctement, vous obtiendrez un résultat similaire à celui qui suit. Votre compartiment cible doit contenir un fichier au format de nom *<your_test_file>*`_encrypted.pdf`, où `<your_test_file>` est le nom du fichier que vous avez chargé.

```
{
    "Contents": [
        {
            "Key": "test_encrypted.pdf",
            "LastModified": "2023-06-07T00:15:50+00:00",
            "ETag": "\"7781a43e765a8301713f533d70968a1e\"",
            "Size": 2763,
            "StorageClass": "STANDARD"
        }
```

```
    ]
}
```

3. Pour télécharger le fichier que Lambda a enregistré dans votre compartiment de destination, exécutez la commande de la CLI suivante. Remplacez le paramètre `--bucket` par le nom de votre compartiment de destination. Pour le paramètre `--key`, utilisez le nom de fichier *<your_test_file>*`_encrypted.pdf`, où <your_test_file> est le nom du fichier de test que vous avez chargé.

```
aws s3api get-object --bucket amzn-s3-demo-bucket-encrypted --
key test_encrypted.pdf my_encrypted_file.pdf
```

   Cette commande télécharge le fichier dans votre répertoire actuel et l'enregistre sous `my_encrypted_file.pdf`.
4. Vérifiez que vous pouvez ouvrir le fichier téléchargé avec le mot de passe avec lequel votre fonction Lambda l'a protégé (`my-secret-password`).

Test de l'application avec le script automatique

Créez les fichiers suivants dans le répertoire de votre projet :

- `test_pdf_encrypt.py` : un script de test que vous pouvez utiliser pour tester automatiquement votre application
- `pytest.ini` : un fichier de configuration pour le script de test

Développez les sections suivantes pour afficher le code et pour en savoir plus sur le rôle de chaque fichier.

Script de test automatisé

Copiez et collez le code suivant dans un fichier nommé `test_pdf_encrypt.py`. Veillez à remplacer les espaces réservés de nom de compartiment :

- Dans la fonction `test_source_bucket_available`, remplacez `amzn-s3-demo-bucket` par le nom de votre compartiment source.
- Dans la fonction `test_encrypted_file_in_bucket`, remplacez `amzn-s3-demo-bucket-encrypted` par *source-bucket*`-encrypted`, où source-bucket> est le nom de votre compartiment source.

- Dans la fonction `cleanup`, remplacez `amzn-s3-demo-bucket` par le nom de votre compartiment source et remplacez `amzn-s3-demo-bucket-encrypted` par le nom de votre compartiment source.

```
import boto3
import json
import pytest
import time
import os

@pytest.fixture
def lambda_client():
    return boto3.client('lambda')

@pytest.fixture
def s3_client():
    return boto3.client('s3')

@pytest.fixture
def logs_client():
    return boto3.client('logs')

@pytest.fixture(scope='session')
def cleanup():
    # Create a new S3 client for cleanup
    s3_client = boto3.client('s3')

    yield
    # Cleanup code will be executed after all tests have finished

    # Delete test.pdf from the source bucket
    source_bucket = 'amzn-s3-demo-bucket'
    source_file_key = 'test.pdf'
    s3_client.delete_object(Bucket=source_bucket, Key=source_file_key)
    print(f"\nDeleted {source_file_key} from {source_bucket}")

    # Delete test_encrypted.pdf from the destination bucket
    destination_bucket = 'amzn-s3-demo-bucket-encrypted'
    destination_file_key = 'test_encrypted.pdf'
    s3_client.delete_object(Bucket=destination_bucket, Key=destination_file_key)
    print(f"Deleted {destination_file_key} from {destination_bucket}")
```

```
@pytest.mark.order(1)
def test_source_bucket_available(s3_client):
    s3_bucket_name = 'amzn-s3-demo-bucket'
    file_name = 'test.pdf'
    file_path = os.path.join(os.path.dirname(__file__), file_name)

    file_uploaded = False
    try:
        s3_client.upload_file(file_path, s3_bucket_name, file_name)
        file_uploaded = True
    except:
        print("Error: couldn't upload file")

    assert file_uploaded, "Could not upload file to S3 bucket"



@pytest.mark.order(2)
def test_lambda_invoked(logs_client):

    # Wait for a few seconds to make sure the logs are available
    time.sleep(5)

    # Get the latest log stream for the specified log group
    log_streams = logs_client.describe_log_streams(
        logGroupName='/aws/lambda/EncryptPDF',
        orderBy='LastEventTime',
        descending=True,
        limit=1
    )

    latest_log_stream_name = log_streams['logStreams'][0]['logStreamName']

    # Retrieve the log events from the latest log stream
    log_events = logs_client.get_log_events(
        logGroupName='/aws/lambda/EncryptPDF',
        logStreamName=latest_log_stream_name
    )

    success_found = False
    for event in log_events['events']:
        message = json.loads(event['message'])
        status = message.get('record', {}).get('status')
```

```
        if status == 'success':
            success_found = True
            break

    assert success_found, "Lambda function execution did not report 'success' status in
 logs."

@pytest.mark.order(3)
def test_encrypted_file_in_bucket(s3_client):
    # Specify the destination S3 bucket and the expected converted file key
    destination_bucket = 'amzn-s3-demo-bucket-encrypted'
    converted_file_key = 'test_encrypted.pdf'

    try:
        # Attempt to retrieve the metadata of the converted file from the destination
 S3 bucket
        s3_client.head_object(Bucket=destination_bucket, Key=converted_file_key)
    except s3_client.exceptions.ClientError as e:
        # If the file is not found, the test will fail
        pytest.fail(f"Converted file '{converted_file_key}' not found in the
 destination bucket: {str(e)}")

def test_cleanup(cleanup):
    # This test uses the cleanup fixture and will be executed last
    pass
```

Le script de test automatique exécute trois fonctions de test pour confirmer le bon fonctionnement de votre application :

- Le test `test_source_bucket_available` confirme que votre compartiment source a été créé avec succès en y téléchargeant un fichier PDF de test.
- Le test `test_lambda_invoked` interroge le dernier flux du journal CloudWatch Logs de votre fonction afin de confirmer que lorsque vous avez chargé le fichier de test, votre fonction Lambda s'est exécutée et a signalé un succès.
- Le test `test_encrypted_file_in_bucket` confirme que votre compartiment de destination contient le fichier chiffré `test_encrypted.pdf`.

Une fois tous ces tests exécutés, le script exécute une étape de nettoyage supplémentaire pour supprimer les fichiers `test.pdf` et `test_encrypted.pdf` de vos compartiments source et de destination.

Comme pour le AWS SAM modèle, les noms de compartiments spécifiés dans ce fichier sont des espaces réservés. Avant d'exécuter le test, vous devez modifier ce fichier avec les noms de compartiment réels de votre application. Cette étape est expliquée plus en détail dans the section called "Test de l'application avec le script automatique".

Fichier de configuration du script de test

Copiez et collez le code suivant dans un fichier nommé `pytest.ini`.

```
[pytest]
markers =
    order: specify test execution order
```

Cela est nécessaire pour spécifier l'ordre dans lequel les tests du script `test_pdf_encrypt.py` s'exécutent.

Pour exécuter l'exemple, procédez comme suit :

1. Assurez-vous que le module `pytest` est installé dans votre environnement local. Vous pouvez installer `pytest` en exécutant les commandes suivantes :

   ```
   pip install pytest
   ```

2. Enregistrez un fichier PDF nommé `test.pdf` dans le répertoire contenant les fichiers `test_pdf_encrypt.py` et `pytest.ini`.
3. Ouvrez un terminal ou un exécuteur shell et exécutez la commande suivante depuis le répertoire contenant les fichiers de test.

   ```
   pytest -s -v
   ```

   Lorsque le test est terminé, vous devriez obtenir un résultat semblable à celui qui suit :

   ```
   ============================================================= test session starts
    ============================================================
   platform linux -- Python 3.12.2, pytest-7.2.2, pluggy-1.0.0 -- /usr/bin/python3
   cachedir: .pytest_cache
   hypothesis profile 'default' -> database=DirectoryBasedExampleDatabase('/home/
   pdf_encrypt_app/.hypothesis/examples')
   Test order randomisation NOT enabled. Enable with --random-order or --random-order-
   bucket=<bucket_type>
   ```

```
rootdir: /home/pdf_encrypt_app, configfile: pytest.ini
plugins: anyio-3.7.1, hypothesis-6.70.0, localserver-0.7.1, random-order-1.1.0
collected 4 items

test_pdf_encrypt.py::test_source_bucket_available PASSED
test_pdf_encrypt.py::test_lambda_invoked PASSED
test_pdf_encrypt.py::test_encrypted_file_in_bucket PASSED
test_pdf_encrypt.py::test_cleanup PASSED
Deleted test.pdf from amzn-s3-demo-bucket
Deleted test_encrypted.pdf from amzn-s3-demo-bucket-encrypted


============================================================== 4 passed in 7.32s
 ============================================================
```

Étapes suivantes

Maintenant que vous avez créé cet exemple d'application, vous pouvez utiliser le code fourni comme base pour créer d'autres types d'applications de traitement de fichiers. Modifiez le code du fichier `lambda_function.py` pour mettre en œuvre la logique de traitement de fichiers adaptée à votre cas d'utilisation.

De nombreux cas d'utilisation typiques du traitement de fichiers impliquent le traitement d'image. Lorsque vous utilisez Python, les bibliothèques de traitement d'image les plus populaires telles que pillow contiennent généralement des composants C ou C++. Afin de garantir que le package de déploiement de votre fonction est compatible avec l'environnement d'exécution Lambda, il est important d'utiliser le binaire de distribution source adéquat.

Lorsque vous déployez vos ressources avec AWS SAM, vous devez prendre des mesures supplémentaires pour inclure la distribution source appropriée dans votre package de déploiement. Étant donné que les dépendances AWS SAM ne seront pas installées pour une plate-forme différente de celle de votre machine de génération, la spécification de la distribution source (`.whl`fichier) correcte dans votre `requirements.txt` fichier ne fonctionnera pas si votre machine de génération utilise un système d'exploitation ou une architecture différent de l'environnement d'exécution Lambda. Au lieu de cela, vous devriez procéder à l'une des opérations suivantes :

- Utilisez l'option `--use-container` lors de l'exécution de `sam build`. Lorsque vous spécifiez cette option, AWS SAM télécharge une image de base de conteneur compatible avec l'environnement d'exécution Lambda et crée le package de déploiement de votre fonction dans

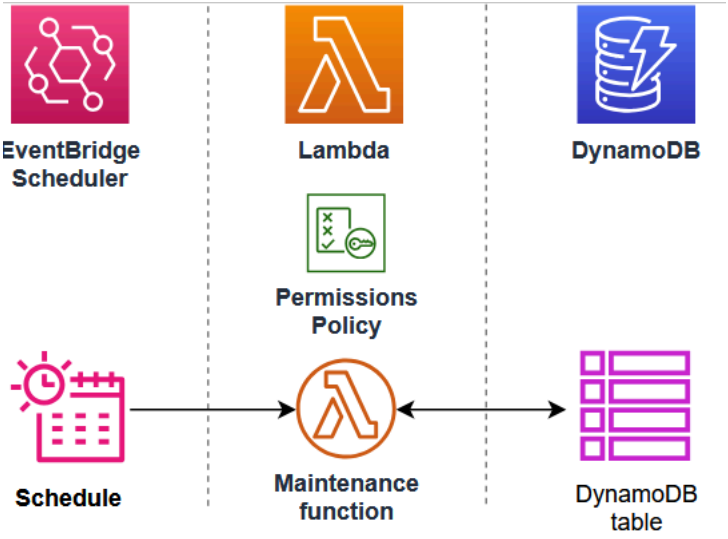
un conteneur Docker à l'aide de cette image. Pour en savoir plus, consultez Building a Lambda function inside of a provided container.

- Créez vous-même le package de déploiement .zip de votre fonction en utilisant le binaire de distribution source approprié et enregistrez le fichier .zip dans le répertoire que vous spécifiez `CodeUri` dans le AWS SAM modèle. Pour en savoir plus sur la création de packages de déploiement .zip pour Python à l'aide de distributions binaires, consultez the section called "Création d'un package de déploiement .zip avec dépendances" et the section called "Création de packages de déploiement .zip avec des bibliothèques natives".

Création d'une application de maintenance planifiée de la base de données

Vous pouvez l'utiliser AWS Lambda pour remplacer les processus planifiés tels que les sauvegardes automatisées du système, les conversions de fichiers et les tâches de maintenance. Dans cet exemple, vous créez une application sans serveur qui effectue une maintenance planifiée régulière sur une table DynamoDB en supprimant les anciennes entrées. L'application utilise EventBridge Scheduler pour appeler une fonction Lambda selon un calendrier cron. Lorsqu'elle est invoquée, la fonction rechercher dans la table pour les éléments datant de plus d'un an et les supprime. La fonction enregistre chaque élément supprimé dans CloudWatch Logs.

Pour implémenter cet exemple, créez d'abord une table DynamoDB et remplissez-la avec des données de test que votre fonction doit interroger. Créez ensuite une fonction Lambda Python avec un déclencheur EventBridge Scheduler et un rôle d'exécution IAM qui autorise la fonction à lire et à supprimer des éléments de votre table.



> **Tip**
>
> Si vous utilisez Lambda pour la première fois, nous vous recommandons de suivre le tutoriel Création de votre première fonction avant de créer cet exemple d'application.

Vous pouvez déployer votre application manuellement en créant et en configurant des ressources à l'aide du AWS Management Console. Vous pouvez également déployer l'application en utilisant le AWS Serverless Application Model (AWS SAM). AWS SAM est un outil d'infrastructure en tant que code (IaC). Avec l'IaC, vous ne créez pas de ressources manuellement, mais vous les définissez dans le code, puis vous les déployez automatiquement.

Si vous souhaitez en savoir plus sur l'utilisation de Lambda avec l'IaC avant de déployer cet exemple d'application, consultez the section called "Infrastructure en tant que code (IaC)".

Conditions préalables

Avant de créer l'exemple d'application, assurez-vous que les outils et programmes de ligne de commande requis sont installés.

- Python

  Pour remplir la table DynamoDB que vous créez pour tester votre application, cet exemple utilise un script Python et un fichier CSV pour écrire des données dans la table. Assurez-vous que Python 3.8 ou version ultérieure est installé sur votre machine.

- AWS SAM INTERFACE DE LIGNE DE COMMANDE (CLI)

  Si vous souhaitez créer la table DynamoDB et déployer l'exemple d'application à l' AWS SAM aide, vous devez installer la CLI. AWS SAM Suivez les instructions d'installation dans le Guise de l'utilisateur AWS SAM .

- AWS CLI

  Pour utiliser le script Python fourni pour remplir votre table de test, vous devez avoir installé et configuré l' AWS CLI. Cela est dû au fait que le script utilise le AWS SDK pour Python (Boto3), qui doit accéder à vos informations d'identification Gestion des identités et des accès AWS (IAM). Vous devez également l' AWS CLI installer pour déployer des ressources à l'aide de AWS SAM. Installez la CLI en suivant les instructions d'installation du Guide de l'utilisateur AWS Command Line Interface .

- Docker

  Pour déployer l'application à l'aide de Docker AWS SAM, vous devez également être installé sur votre machine de compilation. Suivez les instructions dans Install Docker Engine sur le site Web de documentation de Docker.

Téléchargement des exemples de fichiers d’application

Pour créer la base de données d’exemple et l’application de maintenance planifiée, vous devez créer les fichiers suivants dans le répertoire de votre projet :

Exemples de fichiers de base de données

- `template.yaml`- un AWS SAM modèle que vous pouvez utiliser pour créer la table DynamoDB
- `sample_data.csv` : un fichier CSV contenant des exemples de données à charger dans votre tableau
- `load_sample_data.py` : un script Python qui écrit les données du fichier CSV dans le tableau

Fichiers de l’application de maintenance planifiée

- `lambda_function.py` : le code de fonction Python de la fonction Lambda qui effectue la maintenance de la base de données
- `requirements.txt` : un fichier manifeste définissant les dépendances requises par le code de votre fonction Python
- `template.yaml`- un AWS SAM modèle que vous pouvez utiliser pour déployer l'application

Fichier de test

- `test_app.py` : un script Python qui scanne la table et confirme le bon fonctionnement de votre fonction en affichant tous les enregistrements datant de plus d’un an

Développez les sections suivantes pour afficher le code et pour en savoir plus sur le rôle de chaque fichier dans la création et le test de votre application. Pour créer les fichiers sur votre ordinateur local, copiez et collez le code ci-dessous.

AWS SAM modèle (exemple de table DynamoDB)

Copiez et collez le code suivant dans un fichier nommé `template.yaml`.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: SAM Template for DynamoDB Table with Order_number as Partition Key and
 Date as Sort Key

Resources:
  MyDynamoDBTable:
    Type: AWS::DynamoDB::Table
    DeletionPolicy: Retain
    UpdateReplacePolicy: Retain
    Properties:
      TableName: MyOrderTable
      BillingMode: PAY_PER_REQUEST
      AttributeDefinitions:
        - AttributeName: Order_number
          AttributeType: S
        - AttributeName: Date
          AttributeType: S
      KeySchema:
        - AttributeName: Order_number
          KeyType: HASH
        - AttributeName: Date
          KeyType: RANGE
      SSESpecification:
        SSEEnabled: true
      GlobalSecondaryIndexes:
        - IndexName: Date-index
          KeySchema:
            - AttributeName: Date
              KeyType: HASH
          Projection:
            ProjectionType: ALL
      PointInTimeRecoverySpecification:
        PointInTimeRecoveryEnabled: true

Outputs:
  TableName:
    Description: DynamoDB Table Name
    Value: !Ref MyDynamoDBTable
  TableArn:
```

```
    Description: DynamoDB Table ARN
    Value: !GetAtt MyDynamoDBTable.Arn
```

> **Note**
>
> AWS SAM les modèles utilisent une convention de dénomination standard `detemplate.yaml`. Dans cet exemple, vous disposez de deux fichiers modèles : l'un pour créer la base de données d'exemple et l'autre pour créer l'application elle-même. Enregistrez-les dans des sous-répertoires distincts dans votre dossier de projet.

Ce AWS SAM modèle définit la ressource de table DynamoDB que vous créez pour tester votre application. La table utilise une clé primaire `Order_number` avec une clé de tri `Date`. Pour que votre fonction Lambda puisse rechercher des éléments directement par date, nous définissons également un index secondaire global nommé `Date-index`.

Pour en savoir plus sur la création et la configuration d'une table DynamoDB à l'aide de cette ressource `AWS::DynamoDB::Table`, consultez AWS::DynamoDB::Table dans le Guide de l'utilisateur AWS CloudFormation .

Exemple de fichier de données de base de données

Copiez et collez le code suivant dans un fichier nommé `sample_data.csv`.

```
Date,Order_number,CustomerName,ProductID,Quantity,TotalAmount
2023-09-01,ORD001,Alejandro Rosalez,PROD123,2,199.98
2023-09-01,ORD002,Akua Mansa,PROD456,1,49.99
2023-09-02,ORD003,Ana Carolina Silva,PROD789,3,149.97
2023-09-03,ORD004,Arnav Desai,PROD123,1,99.99
2023-10-01,ORD005,Carlos Salazar,PROD456,2,99.98
2023-10-02,ORD006,Diego Ramirez,PROD789,1,49.99
2023-10-03,ORD007,Efua Owusu,PROD123,4,399.96
2023-10-04,ORD008,John Stiles,PROD456,2,99.98
2023-10-05,ORD009,Jorge Souza,PROD789,3,149.97
2023-10-06,ORD010,Kwaku Mensah,PROD123,1,99.99
2023-11-01,ORD011,Li Juan,PROD456,5,249.95
2023-11-02,ORD012,Marcia Oliveria,PROD789,2,99.98
2023-11-03,ORD013,Maria Garcia,PROD123,3,299.97
2023-11-04,ORD014,Martha Rivera,PROD456,1,49.99
2023-11-05,ORD015,Mary Major,PROD789,4,199.96
2023-12-01,ORD016,Mateo Jackson,PROD123,2,199.99
```

```
2023-12-02,ORD017,Nikki Wolf,PROD456,3,149.97
2023-12-03,ORD018,Pat Candella,PROD789,1,49.99
2023-12-04,ORD019,Paulo Santos,PROD123,5,499.95
2023-12-05,ORD020,Richard Roe,PROD456,2,99.98
2024-01-01,ORD021,Saanvi Sarkar,PROD789,3,149.97
2024-01-02,ORD022,Shirley Rodriguez,PROD123,1,99.99
2024-01-03,ORD023,Sofia Martinez,PROD456,4,199.96
2024-01-04,ORD024,Terry Whitlock,PROD789,2,99.98
2024-01-05,ORD025,Wang Xiulan,PROD123,3,299.97
```

Ce fichier contient des exemples de données de test à utiliser dans votre table DynamoDB au format CSV (valeurs séparées par des virgules) standard.

Script Python de chargement d'exemples de données

Copiez et collez le code suivant dans un fichier nommé `load_sample_data.py`.

```
import boto3
import csv
from decimal import Decimal

# Initialize the DynamoDB client
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('MyOrderTable')
print("DDB client initialized.")

def load_data_from_csv(filename):
    with open(filename, 'r') as file:
        csv_reader = csv.DictReader(file)
        for row in csv_reader:
            item = {
                'Order_number': row['Order_number'],
                'Date': row['Date'],
                'CustomerName': row['CustomerName'],
                'ProductID': row['ProductID'],
                'Quantity': int(row['Quantity']),
                'TotalAmount': Decimal(str(row['TotalAmount']))
            }
            table.put_item(Item=item)
            print(f"Added item: {item['Order_number']} - {item['Date']}")

if __name__ == "__main__":
    load_data_from_csv('sample_data.csv')
```

```
    print("Data loading completed.")
```

Ce script Python utilise d'abord le AWS SDK pour Python (Boto3) pour créer une connexion à votre table DynamoDB. Il effectue ensuite une itération sur chaque ligne du fichier CSV de données d'exemple, crée un élément à partir de cette ligne et écrit l'élément dans la table DynamoDB à l'aide du kit SDK boto3.

Code de fonction Python

Copiez et collez le code suivant dans un fichier nommé `lambda_function.py`.

```
import boto3
from datetime import datetime, timedelta
from boto3.dynamodb.conditions import Key, Attr
import logging

logger = logging.getLogger()
logger.setLevel("INFO")

def lambda_handler(event, context):
    # Initialize the DynamoDB client
    dynamodb = boto3.resource('dynamodb')

    # Specify the table name
    table_name = 'MyOrderTable'
    table = dynamodb.Table(table_name)

    # Get today's date
    today = datetime.now()

    # Calculate the date one year ago
    one_year_ago = (today - timedelta(days=365)).strftime('%Y-%m-%d')

    # Scan the table using a global secondary index
    response = table.scan(
        IndexName='Date-index',
        FilterExpression='#date < :one_year_ago',
        ExpressionAttributeNames={
            '#date': 'Date'
        },
        ExpressionAttributeValues={
            ':one_year_ago': one_year_ago
        }
```

```
    )

     # Delete old items
    with table.batch_writer() as batch:
        for item in response['Items']:
            Order_number = item['Order_number']
            batch.delete_item(
                Key={
                    'Order_number': Order_number,
                    'Date': item['Date']
                }
            )
            logger.info(f'deleted order number {Order_number}')

    # Check if there are more items to scan
    while 'LastEvaluatedKey' in response:
        response = table.scan(
            IndexName='DateIndex',
            FilterExpression='#date < :one_year_ago',
            ExpressionAttributeNames={
                '#date': 'Date'
            },
            ExpressionAttributeValues={
                ':one_year_ago': one_year_ago
            },
            ExclusiveStartKey=response['LastEvaluatedKey']
        )

        # Delete old items
        with table.batch_writer() as batch:
            for item in response['Items']:
                batch.delete_item(
                    Key={
                        'Order_number': item['Order_number'],
                        'Date': item['Date']
                    }
                )

    return {
        'statusCode': 200,
        'body': 'Cleanup completed successfully'
    }
```

Le code de la fonction Python contient la fonction de gestion (`lambda_handler`) que Lambda exécute lorsque votre fonction est appelée.

Lorsque la fonction est invoquée par le EventBridge planificateur, il utilise le AWS SDK pour Python (Boto3) pour créer une connexion à la table DynamoDB sur laquelle la tâche de maintenance planifiée doit être effectuée. Il utilise ensuite la bibliothèque Python `datetime` pour calculer la date d'il y a un an, avant de rechercher dans la table les éléments plus anciens et de les supprimer.

Notez que les réponses issues des opérations de requête et d'analyse DynamoDB sont limitées à une taille d'au maximum 1 Mo. Si la réponse est supérieure à 1 Mo, DynamoDB pagine les données et renvoie un élément `LastEvaluatedKey` dans la réponse. Pour garantir que notre fonction traite tous les enregistrements de la table, nous vérifions la présence de cette clé et continuons à analyser la table à partir de la dernière position évaluée jusqu'à ce que toute la table ait été analysée.

Fichier manifeste **`requirements.txt`**

Copiez et collez le code suivant dans un fichier nommé `requirements.txt`.

```
boto3
```

Dans cet exemple, le code de votre fonction n'a qu'une seule dépendance qui ne fait pas partie de la bibliothèque Python standard : le kit SDK pour Python (Boto3) que la fonction utilise pour analyser et supprimer des éléments de la table DynamoDB.

> **Note**
>
> Une version du kit SDK pour Python (Boto3) est incluse dans l'environnement d'exécution Lambda, de sorte que votre code s'exécute sans ajouter Boto3 au package de déploiement de votre fonction. Toutefois, pour garder le contrôle total des dépendances de votre fonction et éviter d'éventuels problèmes de désalignement de version, la bonne pratique pour Python consiste à inclure toutes les dépendances de fonction dans le package de déploiement de votre fonction. Pour en savoir plus, consultez the section called "Dépendances d'exécution dans Python".

AWS SAM modèle (application de maintenance planifiée)

Copiez et collez le code suivant dans un fichier nommé `template.yaml`.

```
AWSTemplateFormatVersion: '2010-09-09'
```

```
Transform: AWS::Serverless-2016-10-31
Description: SAM Template for Lambda function and EventBridge Scheduler rule

Resources:
  MyLambdaFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: ScheduledDBMaintenance
      CodeUri: ./
      Handler: lambda_function.lambda_handler
      Runtime: python3.11
      Architectures:
        - x86_64
      Events:
        ScheduleEvent:
          Type: ScheduleV2
          Properties:
            ScheduleExpression: cron(0 3 1 * ? *)
            Description: Run on the first day of every month at 03:00 AM
      Policies:
        - CloudWatchLogsFullAccess
        - Statement:
            - Effect: Allow
              Action:
                - dynamodb:Scan
                - dynamodb:BatchWriteItem
              Resource: !Sub 'arn:aws:dynamodb:${AWS::Region}:${AWS::AccountId}:table/
MyOrderTable'

  LambdaLogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub /aws/lambda/${MyLambdaFunction}
      RetentionInDays: 30

Outputs:
  LambdaFunctionName:
    Description: Lambda Function Name
    Value: !Ref MyLambdaFunction
  LambdaFunctionArn:
    Description: Lambda Function ARN
    Value: !GetAtt MyLambdaFunction.Arn
```

> **Note**
>
> AWS SAM les modèles utilisent une convention de dénomination standard `detemplate.yaml`. Dans cet exemple, vous disposez de deux fichiers modèles : l'un pour créer la base de données d'exemple et l'autre pour créer l'application elle-même. Enregistrez-les dans des sous-répertoires distincts dans votre dossier de projet.

Ce AWS SAM modèle définit les ressources de votre application. Nous définissons la fonction Lambda à l'aide de la ressource `AWS::Serverless::Function`. Le calendrier du EventBridge planificateur et le déclencheur permettant d'appeler la fonction Lambda sont créés à l'aide de la `Events` propriété de cette ressource à l'aide d'un type de. `ScheduleV2` Pour en savoir plus sur la définition des plannings du EventBridge planificateur dans les AWS SAM modèles, consultez ScheduleV2 dans le manuel du développeur.AWS Serverless Application Model

Outre la fonction Lambda et le calendrier du EventBridge planificateur, nous définissons également un groupe de CloudWatch journaux auquel votre fonction doit envoyer les enregistrements des éléments supprimés.

Script de test

Copiez et collez le code suivant dans un fichier nommé `test_app.py`.

```
import boto3
from datetime import datetime, timedelta
import json

# Initialize the DynamoDB client
dynamodb = boto3.resource('dynamodb')

# Specify your table name
table_name = 'YourTableName'
table = dynamodb.Table(table_name)

# Get the current date
current_date = datetime.now()

# Calculate the date one year ago
one_year_ago = current_date - timedelta(days=365)
```

```
# Convert the date to string format (assuming the date in DynamoDB is stored as a
 string)
one_year_ago_str = one_year_ago.strftime('%Y-%m-%d')

# Scan the table
response = table.scan(
    FilterExpression='#date < :one_year_ago',
    ExpressionAttributeNames={
        '#date': 'Date'
    },
    ExpressionAttributeValues={
        ':one_year_ago': one_year_ago_str
    }
)

# Process the results
old_records = response['Items']

# Continue scanning if we have more items (pagination)
while 'LastEvaluatedKey' in response:
    response = table.scan(
        FilterExpression='#date < :one_year_ago',
        ExpressionAttributeNames={
            '#date': 'Date'
        },
        ExpressionAttributeValues={
            ':one_year_ago': one_year_ago_str
        },
        ExclusiveStartKey=response['LastEvaluatedKey']
    )
    old_records.extend(response['Items'])

for record in old_records:
    print(json.dumps(record))

# The total number of old records should be zero.
print(f"Total number of old records: {len(old_records)}")
```

Ce script de test utilise le AWS SDK pour Python (Boto3) pour créer une connexion à votre table DynamoDB et rechercher les éléments datant de plus d'un an. Pour confirmer que la fonction Lambda s’est bien exécutée, à la fin du test, la fonction affiche le nombre d’enregistrements de plus d’un an encore présents dans la table. Si l’exécution de la fonction Lambda a réussi, le nombre d’anciens enregistrements dans la table doit être égal à zéro.

Création et remplissage de l’exemple de table DynamoDB

Pour tester votre application de maintenance planifiée, vous devez d’abord créer une table DynamoDB et la remplir avec des exemples de données. Vous pouvez créer le tableau manuellement à l’aide de l’ AWS Management Console ou en utilisant AWS SAM. Nous vous recommandons de l'utiliser AWS SAM pour créer et configurer rapidement le tableau à l'aide de quelques AWS CLI commandes.

Console

Créer le tableau DynamoDB

1. Ouvrez la page Tables (Tables) de la console DynamoDB.
2. Choisissez Créer un tableau.
3. Créez la table en procédant comme suit :
   a. Sous Détails de la table, dans Nom de la table, saisissez **`MyOrderTable`**.
   b. Pour Clé de partition, saisissez **`Order_number`**, et conservez le type défini sur Chaîne.
   c. Pour la Clé de tri, saisissez **`Date`** et conservez le type défini sur Chaîne.
   d. Laissez Paramètres du tableau définis sur Paramètres par défaut et choisissez Créer une table.
4. Lorsque la création de votre table est terminée et que son Statut affiche Actif, créez un index secondaire global (GSI) en procédant comme suit. Votre application utilisera ce GSI pour rechercher des éléments directement par date afin de déterminer ceux à supprimer.
   a. MyOrderTableChoisissez dans la liste des tables.
   b. Choisissez l’onglet Index.
   c. Sous Index secondaires globaux, choisissez Créer un index.
   d. Sous Détails de l’index, saisissez **`Date`** la Clé de partition et laissez Type de données défini sur Chaîne.
   e. Pour Nom de l'index, saisissez **`Date-index`**.
   f. Laissez les valeurs par défaut de tous les autres paramètres, faites défiler l’écran vers le bas et choisissez Créer un index.

AWS SAM

Créer le tableau DynamoDB

1. Accédez au dossier dans lequel vous avez enregistré le fichier `template.yaml` de la table DynamoDB. Notez que cet exemple utilise deux fichiers `template.yaml`. Assurez-vous qu'ils sont enregistrés dans des sous-dossiers distincts et que vous vous trouvez dans le bon dossier contenant le modèle pour créer votre table DynamoDB.
2. Exécutez la commande suivante.

   ```
   sam build
   ```

   Cette commande rassemble les artefacts de création pour les ressources que vous souhaitez déployer et les place dans le format et l'emplacement appropriés pour les déployer.
3. Pour créer la ressource DynamoDB spécifiée dans le fichier `template.yaml`, exécutez la commande suivante.

   ```
   sam deploy --guided
   ```

   L'utilisation du `--guided` drapeau signifie que des instructions vous AWS SAM seront affichées pour vous guider tout au long du processus de déploiement. Pour ce déploiement, définissez `Stack name` sur **`cron-app-test-db`** et acceptez les valeurs par défaut pour toutes les autres options à l'aide de la touche Entrée.

   Une AWS SAM fois la création de la ressource DynamoDB terminée, le message suivant devrait s'afficher.

   ```
   Successfully created/updated stack - cron-app-test-db in us-west-2
   ```
4. Vous pouvez également vérifier que la table DynamoDB a été créée en ouvrant la page Tables de la console DynamoDB. Vous devriez voir une table nommée `MyOrderTable`.

Après avoir créé votre table, vous devez ensuite ajouter des exemples de données pour tester votre application. Le fichier CSV `sample_data.csv` que vous avez téléchargé précédemment contient un certain nombre d'exemples d'entrées comprenant des numéros de commande, des dates et des informations sur les clients et les commandes. Utilisez le script python `load_sample_data.py` fourni pour ajouter ces données à votre table.

Pour ajouter les exemples de données à la table

1. Naviguez jusqu'au répertoire contenant les fichiers `sample_data.csv` et `load_sample_data.py`. Si ces fichiers se trouvent dans des répertoires distincts, déplacez-les afin qu'ils soient enregistrés au même endroit.
2. Créez un environnement virtuel Python dans lequel exécuter le script en exécutant la commande suivante. Nous vous recommandons d'utiliser un environnement virtuel, car lors de l'étape suivante, vous devrez installer l' AWS SDK pour Python (Boto3).

   ```
   python -m venv venv
   ```

3. Activez l'environnement virtuel en exécutant la commande suivante.

   ```
   source venv/bin/activate
   ```

4. Installez le kit SDK for Python (Boto3) dans votre environnement virtuel en exécutant la commande suivante. Le script utilise cette bibliothèque pour se connecter à votre table DynamoDB et ajouter les éléments.

   ```
   pip install boto3
   ```

5. Exécutez le script pour remplir la table en exécutant la commande suivante.

   ```
   python load_sample_data.py
   ```

   Si le script s'exécute correctement, il doit afficher chaque élément sur la console au fur et à mesure du chargement et du rapport `Data loading completed`.
6. Désactivez l'environnement virtuel en exécutant la commande suivante.

   ```
   deactivate
   ```

7. Vous pouvez vérifier que les données ont été chargées dans votre table DynamoDB en procédant comme suit :

   a. Ouvrez la page Explorer les éléments de la console DynamoDB et sélectionnez votre table (`MyOrderTable`).

   b. Dans le volet Éléments renvoyés, vous devriez voir les 25 éléments du fichier CSV que le script a ajouté à la table.

Création de l’application de maintenance planifiée

Vous pouvez créer et déployer les ressources pour cet exemple d'application étape par étape en utilisant AWS Management Console ou en utilisant AWS SAM. Dans un environnement de production, nous vous recommandons d'utiliser un outil Infrustracture-as-Code (iAc) permettant de déployer de manière répétitive des applications sans serveur sans recourir AWS SAM à des processus manuels.

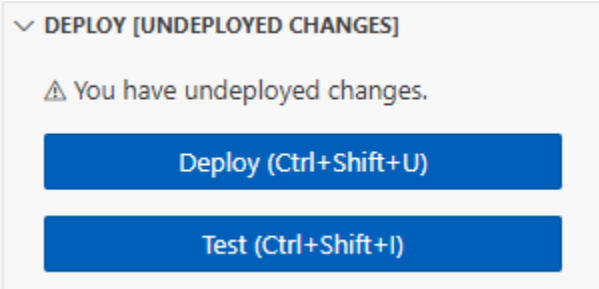
Pour cet exemple, suivez les instructions de la console pour savoir comment configurer chaque AWS ressource séparément, ou suivez les AWS SAM instructions pour déployer rapidement l'application à l'aide de AWS CLI commandes.

Console

Pour créer la fonction à l'aide du AWS Management Console

Créez d’abord une fonction contenant le code de démarrage de base. Vous remplacez ensuite ce code par votre propre code de fonction en copiant et collant le code directement dans l’éditeur de code Lambda ou en chargeant votre code en tant que package `.zip`. Pour cette tâche, nous vous recommandons de copier-coller le code.

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez Créer une fonction.
3. Choisissez Créer à partir de zéro.
4. Sous Informations de base, procédez comme suit :
    a. Sous Nom de la fonction, saisissez **`ScheduledDBMaintenance`**.
    b. Pour Environnement d’exécution, choisissez la dernière version de Python.
    c. Pour Architecture, choisissez x86_64.
5. Choisissez Créer une fonction.
6. Une fois votre fonction créée, vous pouvez la configurer à l’aide du code de fonction fourni.
    a. Dans le volet Source du code, remplacez le code Hello world créé par Lambda par le code de fonction Python du fichier `lambda_function.py` que vous avez enregistré précédemment.
    b. Dans la section DÉPLOYER, choisissez Déployer pour mettre à jour le code de votre fonction :



Pour configurer la mémoire et le délai d'expiration d'une fonction (console)

1. Sélectionnez l'onglet Configuration correspondant à votre fonction.
2. Dans le volet de Configuration générale, choisissez Modifier.
3. Réglez Mémoire sur 256 Mo et Délai d'expiration sur 15 secondes. Si vous traitez une table volumineuse contenant de nombreux enregistrements, par exemple dans le cas d'un environnement de production, vous pouvez envisager de définir un délai d'expiration plus élevé. Cela donne à votre fonction plus de temps pour scanner et nettoyer la base de données.
4. Choisissez Enregistrer.

Pour configurer le format de journal (console)

Vous pouvez configurer les fonctions Lambda pour générer des journaux au format texte non structuré ou au format JSON. Nous vous recommandons d'utiliser le format JSON pour les journaux afin de faciliter la recherche et le filtrage des données de journal. Pour en savoir plus sur les options de configuration du journal Lambda, consultez the section called "Configuration de commandes de journalisation avancées pour votre fonction Lambda".

1. Sélectionnez l'onglet Configuration correspondant à votre fonction.
2. Sélectionnez Outils de surveillance et d'exploitation.
3. Dans le volet de configuration de la journalisation, choisissez Modifier.
4. Pour Configuration de journalisation, sélectionnez JSON.
5. Choisissez Enregistrer.

Pour configurer les autorisations IAM

Pour donner à votre fonction les autorisations nécessaires pour lire et supprimer des éléments DynamoDB, vous devez ajouter une politique au rôle d'exécution de votre fonction définissant les autorisations nécessaires.

1. Ouvrez l'onglet Configuration, puis choisissez Autorisations dans la barre de navigation de gauche.
2. Sous Rôle d'exécution, choisissez le nom du rôle.
3. Dans la console IAM, choisissez Ajouter des autorisations, puis Créer une politique intégrée.
4. Utilisez l'éditeur JSON et saisissez la politique suivante :

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:Scan",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/MyOrderTable"
        }
    ]
}
```

5. Nommez la politique **`DynamoDBCleanupPolicy`**, puis créez-la.

Pour configurer le EventBridge planificateur en tant que déclencheur (console)

1. Ouvrez la EventBridge console.
2. Dans le volet de navigation de gauche, choisissez Planificateurs dans la section Planificateur.
3. Choisissez Créer une planification.
4. Configurez la planification en procédant comme suit :

    a. Pour Nom de la planification, saisissez un nom à attribuer à votre planification (par exemple, **`DynamoDBCleanupSchedule`**).

b. Sous Modèle de planification, choisissez Planification récurrente.

c. Pour Type de planification, laissez la valeur sur Planification basée sur Cron par défaut, puis saisissez les détails de planification suivants :

- Minutes : **0**
- Heures : **3**
- Jour du mois : **1**
- Mois : *****
- Jour de la semaine : **?**
- Année : *****

Lorsqu'elle est évaluée, cette expression cron s'exécute le premier jour de chaque mois à 03h00.

d. Pour Fenêtre horaire flexible, sélectionnez Désactivé.

5. Choisissez Suivant.

6. Configurez le déclencheur de votre fonction Lambda en procédant comme suit :

a. Dans le volet Détails de la cible, laissez API de la cible définie sur Cibles modélisées, puis sélectionnez Invocation AWS Lambda .

b. Sous Invocation, sélectionnez votre fonction Lambda (`ScheduledDBMaintenance`) dans la liste déroulante.

c. Laissez Données utiles vide et choisissez Suivant.

d. Faites défiler la page jusqu'à Autorisations et sélectionnez Créer un rôle pour cette planification. Lorsque vous créez un nouveau calendrier du EventBridge planificateur à l'aide de la console, le EventBridge planificateur crée une nouvelle politique avec les autorisations requises dont le calendrier a besoin pour appeler votre fonction. Pour plus d'informations sur la gestion de vos autorisations de planification, consultez la section Programmations basées sur Cron dans le guide de l'utilisateur du EventBridge planificateur.

e. Choisissez Suivant.

7. Vérifiez vos paramètres et choisissez Créer une planification pour terminer la création de la planification et du déclencheur Lambda.

AWS SAM

Pour déployer l'application à l'aide de AWS SAM

1. Accédez au dossier dans lequel vous avez enregistré le fichier `template.yaml` de l'application. Notez que cet exemple utilise deux fichiers `template.yaml`. Assurez-vous qu'ils sont enregistrés dans des sous-dossiers distincts et que vous vous trouvez dans le bon dossier contenant le modèle pour créer l'application.
2. Copiez les fichiers `lambda_function.py` et `requirements.txt` que vous avez téléchargés précédemment dans le même dossier. L'emplacement du code spécifié dans le AWS SAM modèle est`./`, c'est-à-dire l'emplacement actuel. AWS SAM recherchera dans ce dossier le code de la fonction Lambda lorsque vous tenterez de déployer l'application.
3. Exécutez la commande suivante.

   ```
   sam build --use-container
   ```

   Cette commande rassemble les artefacts de création pour les ressources que vous souhaitez déployer et les place dans le format et l'emplacement appropriés pour les déployer. En spécifiant l'option `--use-container`, vous créez votre fonction à l'intérieur d'un conteneur Docker de type Lambda. Nous l'utilisons ici, vous n'avez donc pas besoin d'installer Python 3.12 sur votre machine locale pour que la compilation fonctionne.
4. Pour créer les ressources Lambda et EventBridge Scheduler spécifiées dans le `template.yaml` fichier, exécutez la commande suivante.

   ```
   sam deploy --guided
   ```

   L'utilisation du `--guided` drapeau signifie que des instructions vous AWS SAM seront affichées pour vous guider tout au long du processus de déploiement. Pour ce déploiement, définissez `Stack name` sur **`cron-maintenance-app`** et acceptez les valeurs par défaut pour toutes les autres options à l'aide de la touche Entrée.

   Lorsque vous aurez AWS SAM terminé de créer les ressources Lambda et EventBridge Scheduler, le message suivant devrait s'afficher.

   ```
   Successfully created/updated stack - cron-maintenance-app in us-west-2
   ```

5. Vous pouvez également vérifier que la fonction Lambda a été créée en ouvrant la page Fonctions de la console Lambda. Vous devriez voir une fonction nommée `ScheduledDBMaintenance`.

Test de l’application

Pour vérifier que votre planification déclenche correctement votre fonction et que celle-ci nettoie correctement les enregistrements de la base de données, vous pouvez modifier temporairement votre planification pour qu’elle ne s’exécute qu’une seule fois à une heure précise. Vous pouvez ensuite exécuter à nouveau `sam deploy` pour réinitialiser votre planification récurrente afin qu’elle s’exécute une fois par mois.

Pour exécuter l'application à l'aide du AWS Management Console

1. Retournez à la page de la console du EventBridge planificateur.
2. Choisissez votre planification, puis choisissez Modifier.
3. Dans la section Modèle de planification, sous Récurrence, sélectionnez Planification ponctuelle.
4. Réglez votre durée d’invocation sur quelques minutes, vérifiez vos paramètres, puis choisissez Enregistrer.

Une fois la planification exécutée et sa cible invoquée, vous exécutez le script `test_app.py` pour vérifier que votre fonction a bien supprimé tous les anciens enregistrements de la table DynamoDB.

Pour vérifier que les anciens enregistrements sont supprimés à l’aide d’un script Python

1. Dans votre ligne de commande, accédez au dossier dans lequel vous avez enregistré `test_app.py`.
2. Exécutez le script.

   ```
   python test_app.py
   ```

   En cas de succès, vous verrez la sortie suivante.

   ```
   Total number of old records: 0
   ```

Étapes suivantes

Vous pouvez désormais modifier le calendrier du EventBridge planificateur pour répondre aux exigences spécifiques de votre application. EventBridge Le planificateur prend en charge les expressions de planification suivantes : cron, rate et plannings ponctuels.

Pour plus d'informations sur les expressions de planification du EventBridge planificateur, consultez la section Types de planification dans le guide de l'utilisateur du EventBridge planificateur. Gestion des accès dans le guide de l'utilisateur IAM

Création d'un système de traitement des commandes avec des fonctions Lambda Durable

> **Note**
>
> BESOIN : Ajouter un schéma d'architecture illustrant API Gateway, le flux de travail Durable Function et les services de support (DynamoDB,) EventBridge

Conditions préalables

- AWS CLI installé et configuré
- BESOIN : Exigences spécifiques relatives aux fonctions durables

Création des fichiers de code source

Créez les fichiers suivants dans le répertoire de votre projet :

- `lambda_function.py`- le code de fonction
- `requirements.txt`- manifeste des dépendances

Code de fonction

```
# NEED: Verify correct imports
import boto3
import json
```

```
def lambda_handler(event, context):
    # NEED: Verify DurableContext syntax
    durable = context.durable

    try:
        # Validate and store order
        order = await durable.step('validate', async () => {
            return validate_order(event['order'])
        })

        # Process payment
        # NEED: Verify wait syntax
        await durable.wait(/* wait configuration */)

        # Additional steps
        # NEED: Complete implementation

    except Exception as e:
        # NEED: Error handling patterns
        raise e

def validate_order(order_data):
    # NEED: Implementation
    pass
```

Fichier Requirements

```
# NEED: List of required packages
```

Déployer l'application

Création d'une table DynamoDB pour les commandes

1. Ouvrez la console DynamoDB à l'adresse https://console.aws.amazon.com/dynamodb/
2. Choisissez Create table
3. Dans le champ Nom de la table, entrez **`Orders`**
4. Pour Clé de partition, entrez **`orderId`**
5. Conserver les autres paramètres par défaut
6. Choisissez Create table

Création de la fonction Lambda

1. Ouvrez la console Lambda à l'adresse https://console.aws.amazon.com/lambda/
2. Choisissez Créer une fonction.
3. Sélectionnez l'auteur à partir de zéro
4. Sous Nom de la fonction, saisissez **`ProcessOrder`**.
5. Pour Runtime, choisissez votre environnement d'exécution préféré
6. BESOIN : Ajouter une configuration spécifique à Durable Functions
7. Choisissez Créer une fonction.

Création du point de terminaison API Gateway

1. Ouvrez la console API Gateway à l'adresse https://console.aws.amazon.com/apigateway/
2. Choisissez Create API
3. Sélectionnez l'API HTTP
4. Choisissez Build
5. Ajoutez une intégration avec votre fonction Lambda
6. Configuration des itinéraires pour le traitement des commandes
7. Déploiement de l’API

Vérification de l'application

Soumettre une commande d'essai :

```
{
    "orderId": "12345",
    "items": [
        {
            "productId": "ABC123",
            "quantity": 1
        }
    ]
}
```

BESOIN : Ajouter des instructions de surveillance spécifiques pour les fonctions durables

Étapes suivantes

Ajouter une logique métier

Mettre en œuvre la gestion des stocks :

```
async def check_inventory(order):
    # Add inventory check logic
    pass
```

Ajoutez les calculs de prix :

```
async def calculate_total(order):
    # Add pricing logic
    pass
```

Améliorez la gestion des erreurs

Ajoutez une logique de compensation :

```
async def reverse_payment(order):
    # Add payment reversal logic
    pass
```

Gérer les annulations de commandes :

```
async def cancel_order(order):
    # Add cancellation logic
    pass
```

Intégrer des systèmes externes

```
async def notify_shipping_provider(order):
    # Add shipping integration
    pass

async def send_customer_notification(order):
    # Add notification logic
    pass
```

Améliorez la surveillance

- Création de CloudWatch tableaux de bord
- Configurer des métriques pour les délais de traitement des commandes
- Configuration des alertes pour les commandes retardées

Outils de développement pour Lambda

Vous avez accès à une variété d'outils qui augmentent la productivité et la facilité d'utilisation tout au long du cycle de développement. Cette section propose des informations sur des outils qui aident de nombreux clients Lambda à concevoir, développer et gérer leurs applications. Du développement local dans votre IDE au déploiement et à la gestion d'applications sans serveur complexes, ces outils vous aident à rationaliser votre flux de travail, améliorer la qualité de votre code et accélérer le développement de solutions robustes basées sur Lambda.

- Développement local : écrivez et testez les fonctions Lambda plus rapidement dans votre environnement de développement préféré. AWS Toolkit pour VS Code permet le développement, le débogage et le test de fonctions locales avec des fonctionnalités de déploiement direct sur Lambda.
- Infrastructure en tant que code (IaC) : déployez et gérez les applications sans serveur de manière cohérente, des tests locaux à la production. AWS SAM, AWS CDK et CloudFormation vous permettent de définir et de gérer votre infrastructure sans serveur via du code pour des déploiements cohérents et avec contrôle des versions.
- GitHub Actions : automatisez les déploiements Lambda directement depuis votre référentiel de code. GitHub Actions vous permet de configurer des flux de travail qui déploient automatiquement vos fonctions Lambda chaque fois que vous envoyez (push) du code ou des modifications de configuration, simplifiant ainsi votre pipeline CI/CD.
- Powertools pour AWS Lambda : créez des applications sans serveur prêtes pour la production avec moins de code personnalisé. Powertools pour AWS Lambda (également appelé Powertools pour AWS) est une boîte à outils de développement open source qui simplifie la mise en œuvre des bonnes pratiques sans serveur comme l'observabilité, la récupération de paramètres et l'idempotence en Python, TypeScript, Java et .NET.
- Flux de travail et événements : coordonnez les fonctions Lambda avec des services AWS, des API et des systèmes externes. AWS Step Functions et Amazon EventBridge fournissent des fonctionnalités d'orchestration et de gestion des événements pour coordonner les flux de travail d'applications complexes basés sur Lambda.

Outils de développement locaux

Les environnements de développement locaux vous permettent de travailler hors ligne et de tirer parti des fonctionnalités avancées de l'IDE tout en itérant rapidement vos fonctions Lambda. Ces outils

vous aident à déboguer les fonctions complexes et à les développer dans des environnements où la connectivité est limitée. Ils favorisent également la collaboration en équipe et l'intégration avec les systèmes de contrôle de version.

Pour plus d'informations sur le développement de fonctions Lambda localement, consultez Développement de fonctions Lambda localement avec VS Code. Cette page explique comment déplacer le développement de fonctions Lambda de la console AWS vers Visual Studio Code, qui fournit un environnement de développement riche avec des fonctionnalités comme le débogage et la complétion du code. Pour effectuer la transition, vous devez configurer AWS Toolkit for Visual Studio Code et les informations d'identification, après quoi vous pourrez utiliser les fonctionnalités avancées de VS Code tout en conservant la possibilité de déployer directement sur AWS.

Le développement local pour Lambda offre plusieurs fonctionnalités clés :

- Utiliser l'intégration de Visual Studio Code avec la console Lambda
- Configurer des environnements de développement Lambda locaux
- Déboguer et tester les fonctions localement
- Appliquer les bonnes pratiques de gestion des fonctions locales

Pour de plus amples informations, consultez Développement de fonctions Lambda localement avec VS Code.

Outils d'infrastructure en tant que code (IaC)

Avec les outils d'infrastructure en tant que code (IaC), vous pouvez définir et gérer votre architecture sans serveur à l'aide de code. Cette approche permet de maintenir la cohérence entre les environnements, contrôler les versions de votre infrastructure et faciliter les pratiques DevOps. L'IaC est particulièrement utile pour automatiser les déploiements, garantir la cohérence des environnements et gérer les déploiements multirégionaux.

Les principaux outils et concepts IaC pour Lambda incluent des cadres pour la création de modèles, la gestion du déploiement et des bonnes pratiques pour l'infrastructure sans serveur :

- Principes fondamentaux de l'IaC pour le développement Lambda
- Fonctionnalités d'CloudFormation, AWS SAM et AWS CDK
- Critères de sélection et comparaison des outils

- Bonnes pratiques de mise en œuvre l'IaC Lambda

Que vous travailliez de manière indépendante sur un petit projet ou que vous fassiez partie d'une grande équipe gérant des applications sans serveur à l'échelle de l'entreprise, ces outils de développement et de déploiement peuvent vous aider à écrire, déployer et gérer vos fonctions Lambda de manière plus efficace.

Pour de plus amples informations, consultez Utilisation de Lambda avec infrastructure en tant que code (IaC).

Outils GitHub Actions

GitHub Actions fournit des fonctionnalités de déploiement automatique pour vos fonctions Lambda directement depuis votre référentiel de code. En créant des fichiers de flux de travail dans votre référentiel, vous pouvez déployer automatiquement des fonctions Lambda chaque fois que des modifications de code ou de configuration sont apportées, rationalisant ainsi votre pipeline d'intégration continue et de déploiement continu (CI/CD). L'action Déployer la fonction Lambda propose une interface YAML déclarative qui simplifie le processus de déploiement, gère les informations d'identification AWS via OpenID Connect (OIDC) et prend en charge divers scénarios de déploiement, notamment les mises à jour du code, les modifications de configuration et les validations d'essai. Cette intégration permet aux équipes de maintenir un processus de déploiement cohérent et automatisé, tout en tirant parti de leurs flux de travail GitHub existants.

Pour de plus amples informations, consultez Utilisation de GitHub Actions pour déployer des fonctions Lambda.

Powertools pour AWS Lambda

Powertools pour AWS est une boîte à outils de développement open source qui vous aide à mettre en œuvre les bonnes pratiques sans serveur avec un minimum de code personnalisé. Disponible pour Python, TypeScript/Node.js, Java et .NET, il fournit des fonctions utilitaires, des décorateurs et des intergiciels qui rationalisent les tâches de développement Lambda courantes. La boîte à outils inclut des fonctionnalités d'observabilité intégrées comme la journalisation structurée, le suivi et la collecte de métriques, notamment avec des utilitaires pour la récupération des paramètres, la gestion des secrets et des modèles d'idempotence. Ces outils s'alignent sur les bonnes pratiques AWS Well-Architected et aident les développeurs à créer plus efficacement des applications sans serveur prêtes pour la production. En réduisant le code standard et en normalisant les modèles courants, Powertools

pour AWS permet aux équipes de se concentrer sur la logique métier tout en maintenant une mise en œuvre cohérente des bonnes pratiques sans serveur dans leurs applications.

Pour de plus amples informations, consultez Powertools pour AWS Lambda.

Outils de gestion des flux de travail et des événements

Les applications Lambda peuvent être utilisées pour l'orchestration de flux de travail complexes et la gestion de divers événements. AWS fournit des outils spécialisés pour vous aider à gérer ces aspects du développement sans serveur. Apprenez-en plus sur AWS Step Functions pour l'orchestration des flux de travail et Amazon EventBridge pour la gestion des événements, ainsi que la manière de les intégrer à vos fonctions Lambda. Ces outils peuvent améliorer de manière significative la capacité de mise à l'échelle et la fiabilité de vos applications sans serveur en fournissant une gestion d'état robuste et des architectures pilotées par les événements. En tirant parti de ces services, vous pouvez créer des solutions Lambda plus sophistiquées et résilientes, capables de gérer des processus métier complexes et de réagir à un large éventail d'événements liés au système et aux applications.

Pour de plus amples informations, consultez Gestion des flux de travail et des événements Lambda.

Développement de fonctions Lambda localement avec VS Code

Vous pouvez déplacer vos fonctions Lambda de la console Lambda vers Visual Studio Code, qui fournit un environnement de développement complet et vous permet d'utiliser d'autres options de développement locales, comme AWS SAM et AWS CDK.

Principaux avantages du développement local

Bien que la console Lambda permette de modifier et de tester rapidement des fonctions, le développement local offre des fonctionnalités plus avancées :

- Fonctionnalités avancées de l'IDE : outils de débogage, de complétion de code et de refactorisation
- Développement hors ligne : travaillez et testez les modifications localement avant de les déployer dans le cloud
- Intégration de l'infrastructure en tant que code : utilisation fluide avec AWS SAM, AWS CDK et Infrastructure Composer
- Gestion des dépendances : contrôle total sur les dépendances des fonctions

Prérequis

Avant de développer des fonctions Lambda localement dans VS Code, vous devez disposer de ce qui suit :

- VS Code : pour les instructions d'installation, consultez Télécharger VS Code.
- AWS Toolkit for Visual Studio Code : pour les instructions d'installation, consultez Configuration d'AWS Toolkit for Visual Studio Code. Pour avoir une présentation, consultez AWS Toolkit for Visual Studio Code.
- Informations d'identification AWS : pour plus d'informations sur la configuration des informations d'identification, consultez Configuration de vos informations d'identification AWS.
- AWS SAM CLI : pour les instructions d'installation, consultez Installation d'AWS SAM CLI.
- Docker installé (facultatif, mais obligatoire pour les tests locaux) : pour les instructions d'installation, consultez Obtenir Docker.

> **Note**
>
> Si vous avez déjà configuré un compte AWS et un profil localement, assurez-vous que la politique gérée AdministratorAccess est ajoutée à votre profil AWS configuré.

Authentification et contrôle d'accès

Pour développer des fonctions Lambda localement, vous avez besoin d'informations d'identification AWS pour accéder aux ressources AWS de façon sécurisée et les gérer en votre nom, comme cela se produirait dans le cloud. AWS Toolkit pour VS Code prend en charge les méthodes d'authentification suivantes :

AWS Toolkit pour VS Code prend en charge les méthodes d'authentification suivantes :

- Informations d'identification à long terme des utilisateurs IAM
- Informations d'identification temporaires de rôles assumés
- Fédération des identités
- Informations d'identification d'utilisateur racine du compte AWS (non recommandé)

Cette section vous explique comment obtenir et configurer ces informations d'identification à l'aide des informations d'identification à long terme d'utilisateur IAM.

Obtenir les informations d'identification IAM

Si un utilisateur IAM possède déjà des clés d'accès, préparez l'ID de clé d'accès et la clé d'accès secrète pour la section suivante. Si vous ne possédez pas ces clés, suivez ces étapes pour les créer :

> **Note**
>
> Vous devez utiliser à la fois l'ID de la clé d'accès et la clé d'accès secrète pour authentifier vos demandes.

Pour créer un utilisateur IAM et des clés d'accès :

1. Ouvrez la console IAM à l'adresse https://console.aws.amazon.com/iam.
2. Dans le panneau de navigation, choisissez utilisateurs.
3. Choisissez Create user (Créer un utilisateur).
4. Dans Nom d'utilisateur, entrez un nom et choisissez Suivant.
5. Pour Set permissions (Définir les autorisations), sélectionnez Attacher directement les stratégies.
6. Sélectionnez AdministratorAccess, puis Suivant.
7. Choisissez Create user (Créer un utilisateur).
8. Dans la bannière indiquant le succès, choisissez Voir l'utilisateur.
9. Choisissez Create access key (Créer une clé d'accès).
10. Pour Cas d'utilisation, sélectionnez Code local.
11. Cochez la case de confirmation et choisissez Suivant.
12. (Facultatif) Entrez une valeur de balise de description.
13. Choisissez Create access key (Créer une clé d'accès).
14. Copiez immédiatement votre clé d'accès et votre clé d'accès secrète. Vous ne pourrez plus accéder à la clé d'accès secrète après avoir quitté cette page.

> **Important**
>
> Ne partagez jamais votre clé secrète et ne la confiez jamais au contrôle de source. Stockez ces clés en toute sécurité et supprimez-les lorsqu'elles ne sont plus nécessaires.

> **Note**
>
> Pour plus d'informations, consultez les Créer un utilisateur IAM dans votre compte AWS et Gérer les clés d'accès pour les utilisateurs IAM dans le Guide de l'utilisateur IAM.

Configurer les informations d'identification AWS à l'aide d'AWS Toolkit

Le tableau suivant récapitule le processus de configuration des informations d'identification que vous allez effectuer dans le cadre de la procédure suivante.

Que faire	Pourquoi ?
Ouvrez le panneau Connexion	Démarrer l'authentification
Utilisez la palette de commandes et recherchez Ajouter une nouvelle connexion AWS	Accédez à l'interface de connexion
Choisissez Informations d'identification IAM	Utiliser vos clés d'accès pour l'accès programmatique
Saisissez le nom du profil, la clé d'accès et la clé secrète	Fournir les informations d'identification pour la connexion
Consultez la mise à jour de l'explorateur AWS	Confirmez que vous êtes connecté

Procédez comme suit pour vous authentifier auprès de votre compte AWS :

1. Ouvrez le panneau Connexion dans VS Code :
   a. Pour démarrer le processus d'authentification, sélectionnez l'icône AWS dans le volet de navigation de gauche ou ouvrez la palette de commandes (Cmd+Maj+P sur Mac ou Ctrl+Maj+P sur Windows/Linux), recherchez et sélectionnez Ajouter une nouvelle connexion AWS.

2. Dans le panneau de connexion, choisissez Informations d'identification IAM, puis sélectionnez Continuer.

> **Note**
>
> Pour continuer, vous devez autoriser les extensions IDE AWS pour VS Code à accéder à vos données.

3. Saisissez le nom de votre profil, l'ID de clé d'accès et la clé d'accès secrète, puis sélectionnez Continuer.
4. Vérifiez la connexion en observant l'explorateur AWS dans VS Code pour vos services et ressources AWS.

Pour plus d'informations sur la configuration de l'authentification avec des informations d'identification à long terme, consultez Utilisation d'informations d'identification à long terme pour authentifier les SDK et outils AWS.

Pour plus d'informations sur la configuration de l'authentification, consultez Informations d'identification AWS IAM dans le Guide de l'utilisateur AWS Toolkit for Visual Studio Code.

Déplacement de la console au développement local

> **Note**
>
> Si vous avez apporté des modifications à la console, assurez-vous qu'il n'y a aucune modification non déployée avant de passer au développement local.

Pour déplacer une fonction Lambda de la console Lambda vers VS Code, procédez comme suit :

1. Ouvrez la console Lambda.
2. Choisissez le nom de votre fonction .
3. Sélectionnez l'onglet Source du code.
4. Choisissez Ouvrir dans Visual Studio Code.

> **Note**
>
> Le bouton Ouvrir dans Visual Studio Code n'est disponible que dans les versions 3.69.0 et ultérieures d'AWS Toolkit. Si une version antérieure d'AWS Toolkit est installée, un message `Cannot open the handler` peut s'afficher dans VS Code. Pour résoudre ce problème, mettez à jour AWS Toolkit vers la dernière version.

5. Lorsque vous y êtes invité, autorisez votre navigateur à ouvrir VS Code.

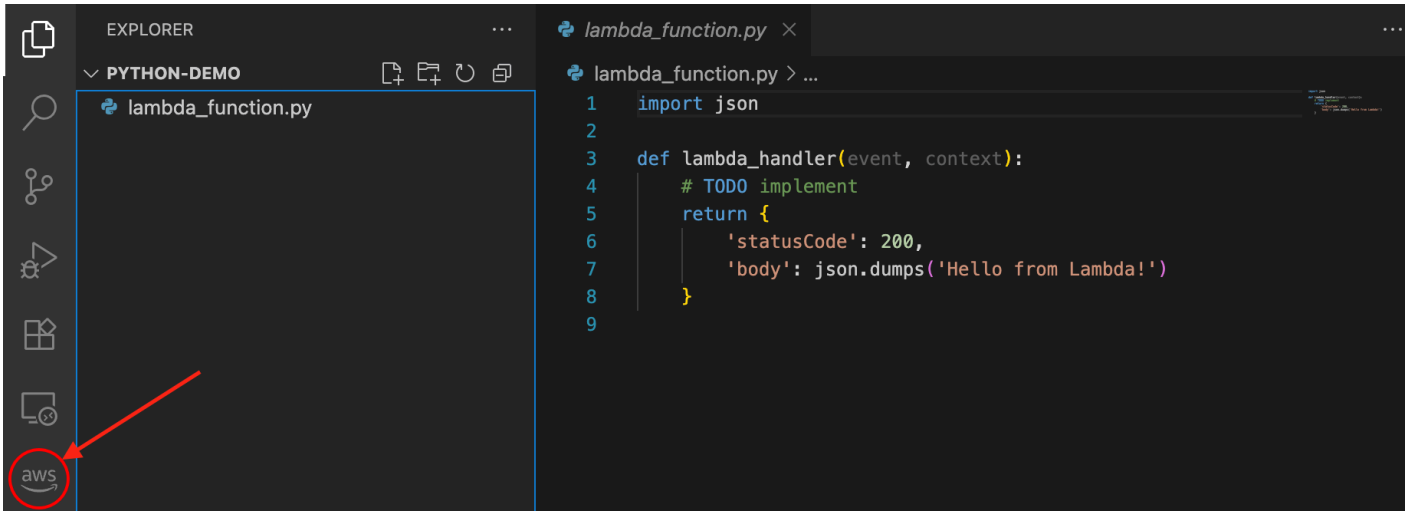
Lorsque vous ouvrez votre fonction dans VS Code, Lambda crée un projet local avec le code de votre fonction dans un emplacement temporaire conçu pour des tests et un déploiement rapides. Cela inclut le code de fonction, les dépendances et une structure de projet de base que vous pouvez utiliser pour le développement local.

Pour plus de détails sur l'utilisation d'AWS dans VS Code, consultez le Guide de l'utilisateur AWS Toolkit for Visual Studio Code.

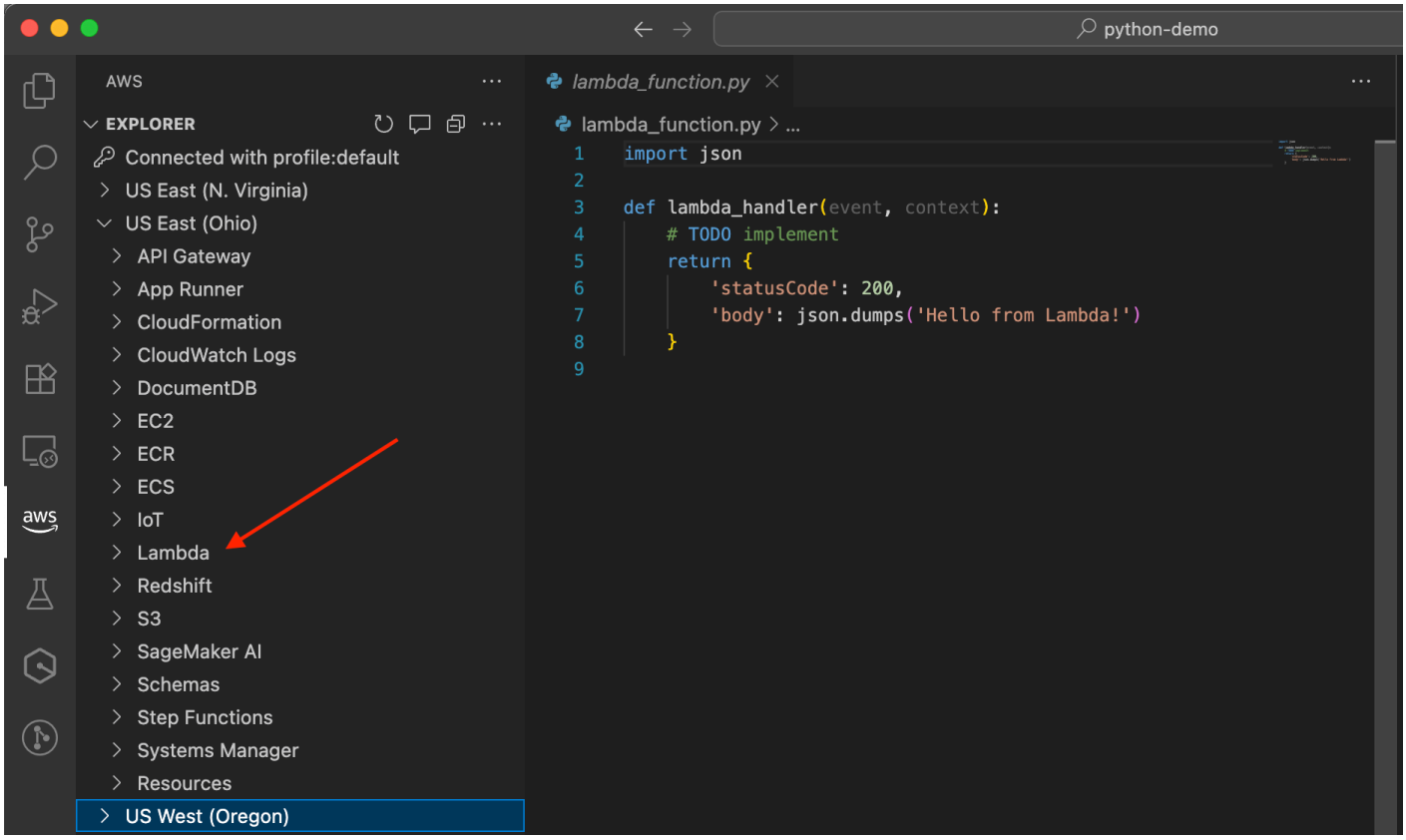
Utilisation des fonctions localement

Après avoir ouvert votre fonction dans VS Code, suivez ces étapes pour accéder à vos fonctions et les gérer :

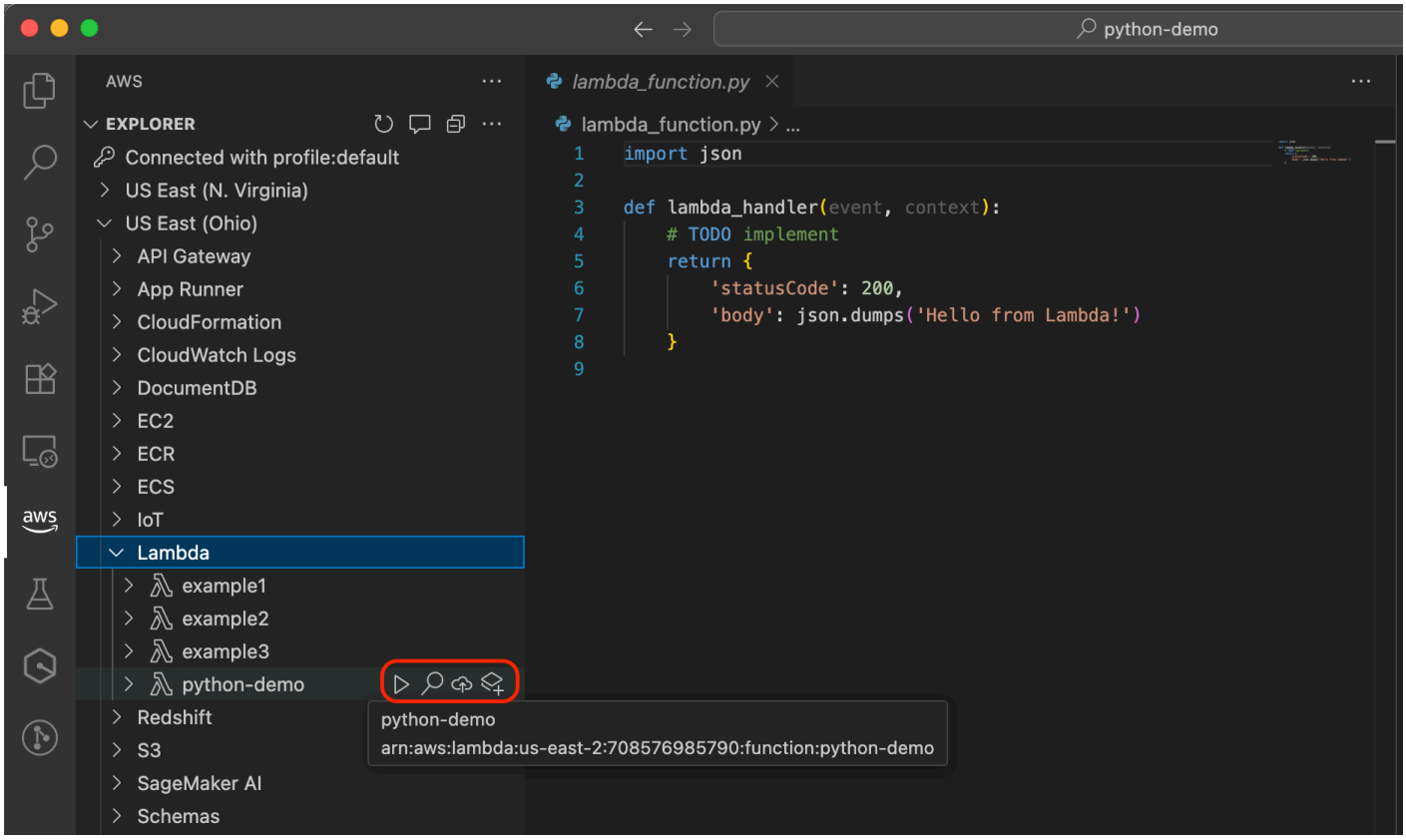
1. Sélectionnez l'icône AWS dans la barre latérale pour ouvrir l'explorateur AWS :



2. Dans l'explorateur AWS, sélectionnez la région dans laquelle se trouve votre fonction Lambda :



3. Dans la région que vous avez sélectionnée, développez la section Lambda pour voir et gérer vos fonctions :



Une fois votre fonction ouverte dans VS Code, vous pouvez :

- Modifier le code de fonction avec prise en charge complète du langage et saisie semi-automatique du code.
- Utilisez l'intégration LocalStack dans VS Code pour tester les fonctions Lambda qui envoient des appels d'API à d'autres Services AWS pendant l'exécution, par exemple en lisant des tables DynamoDB ou en écrivant dans des compartiments Amazon S3. LocalStack est un émulateur de services cloud qui fournit un environnement de développement local complet pour tester les intégrations de service. Vous pouvez également utiliser AWS SAM CLI pour tester votre fonction dans un conteneur local. Si votre fonction fait des appels d'API à d'autres Services AWS, ces appels atteindront des ressources AWS réelles, et non des ressources émulées.
- Déboguez votre fonction avec des points d'arrêt et l'inspection des variables. Pour plus d'informations, consultez Exécution et débogage de fonctions Lambda directement à partir du code dans le Guide de l'utilisateur AWS Toolkit for Visual Studio Code.
- Déployez à nouveau votre fonction mise à jour sur AWS en utilisant l'icône cloud.
- Installez et gérez les dépendances de votre fonction.

Pour plus d'informations, veuillez consulter la section Utilisation des fonctions AWS Lambda dans le guide de l'utilisateur AWS Toolkit for Visual Studio Code.

Convertir votre fonction en modèle AWS SAM et utiliser les outils IaC

Dans VS Code, vous pouvez convertir votre fonction Lambda en modèle AWS SAM en choisissant l'icône Convertir en application AWS SAM à côté de votre fonction Lambda. Vous serez invité à sélectionner l'emplacement d'un projet AWS SAM. Une fois sélectionnée, votre fonction Lambda sera convertie en `template.yaml` fichier enregistré dans votre nouveau projet AWS SAM.

Une fois votre fonction convertie en modèle AWS SAM, vous pouvez :

- Contrôler la gestion des versions de votre infrastructure
- Automatiser les déploiements
- Déboguer les fonctions à distance
- Ajouter des ressources AWS supplémentaires à votre application
- Maintenir des environnements cohérents tout au long de votre cycle de développement
- Utiliser Infrastructure Composer pour modifier visuellement votre modèle AWS SAM

Pour plus d'informations sur l'utilisation des outils IaC, consultez les guides suivants :

- Le Guide du développeur AWS Serverless Application Model
- Le Guide du développeur AWS Cloud Development Kit (AWS CDK)
- Guide du développeur Infrastructure Composer
- Le guide de l'utilisateur du AWS CloudFormation

Ces outils fournissent des fonctionnalités supplémentaires pour définir, tester et déployer vos applications sans serveur.

Étapes suivantes

Pour en savoir plus sur l'utilisation des fonctions Lambda dans VS Code, consultez les ressources suivantes :

- Utilisation des fonctions Lambda AWS dans le Guide de l'utilisateur d'AWS Toolkit pour VS Code
- Utilisation d'applications sans serveur dans le Guide de l'utilisateur d'AWS Toolkit pour VS Code.

- Infrastructure en tant que code dans le Guide du développeur Lambda

Utilisation de GitHub Actions pour déployer des fonctions Lambda

Vous pouvez utiliser GitHub Actions pour déployer automatiquement des fonctions Lambda lorsque vous envoyez des modifications de code ou de configuration à votre référentiel. L'action Déployer la fonction Lambda fournit une interface YAML simple et déclarative qui élimine la complexité des étapes de déploiement manuelles.

Exemple de flux de travail

Pour configurer le déploiement automatique des fonctions Lambda, créez un fichier de flux de travail dans le répertoire `.github/workflows/` de votre référentiel :

Example Flux de travail GitHub Actions pour le déploiement de Lambda

```
name: Deploy AWS Lambda

on:
  push:
    branches:
      - main

jobs:
  deploy:
    runs-on: ubuntu-latest
    permissions:
      id-token: write # Required for OIDC authentication
      contents: read  # Required to check out the repository
    steps:
      - uses: actions/checkout@v4

      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v4
        with:
          role-to-assume: arn:aws:iam::123456789012:role/GitHubActionRole
          aws-region: us-east-1

      - name: Deploy Lambda Function
        uses: aws-actions/aws-lambda-deploy@v1
        with:
```

```
          function-name: my-lambda-function
          code-artifacts-dir: ./dist
```

Ce flux de travail s'exécute lorsque vous apportez des modifications à la branche `main`. Il vérifie votre référentiel, configure les informations d'identification AWS à l'aide d'OpenID Connect (OIDC) et déploie votre fonction à l'aide du code dans le répertoire `./dist`.

Pour des exemples supplémentaires, notamment la mise à jour de la configuration des fonctions, le déploiement via des compartiments S3 et la validation de l'essai, consultez le fichier README de Déployer la fonction Lambda.

Ressources supplémentaires

- Configurer les informations d'identification AWS (GitHub Actions)
- Configuration d'OpenID Connect dans AWS

Utilisation de Lambda avec infrastructure en tant que code (IaC)

Les fonctions Lambda s'exécutent rarement de manière isolée. Elles font souvent partie d'une application sans serveur avec d'autres ressources telles que les bases de données, les files d'attente et le stockage. Avec l'infrastructure en tant que code (IaC), vous pouvez automatiser vos processus de déploiement pour déployer et mettre à jour rapidement et de manière répétée des applications sans serveur complètes impliquant de nombreuses ressources AWS distinctes. Cette approche accélère votre cycle de développement, facilite la gestion des configurations et garantit que vos ressources sont déployées de la même manière à chaque fois.

Outils de l'IaC pour Lambda

CloudFormation

: CloudFormation est le service IaC de base d'AWS. Vous pouvez utiliser des modèles YAML ou JSON pour modéliser et allouer l'ensemble de votre infrastructure AWS, y compris les fonctions Lambda. CloudFormation gère les complexités liées à la création, à la mise à jour et à la suppression de vos ressources AWS.

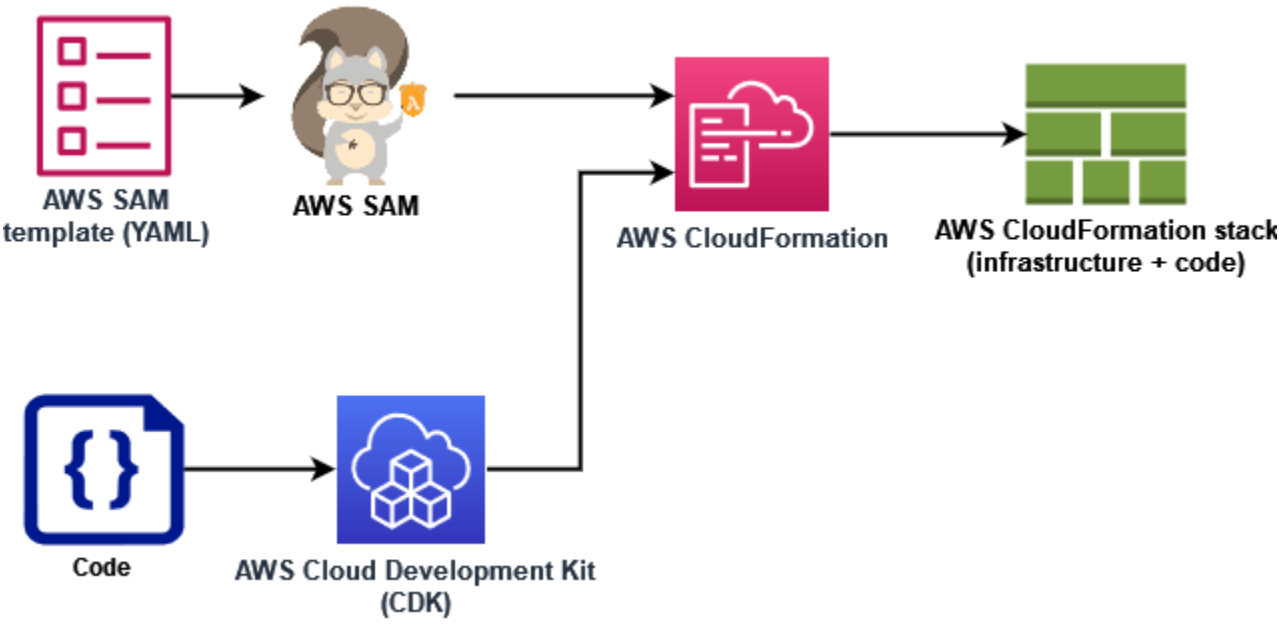
AWS Serverless Application Model (AWS SAM)

: AWS SAM est une infrastructure open source qui repose sur CloudFormation. Il fournit une syntaxe simplifiée pour définir les applications sans serveur. Utilisez des modèles AWS SAM pour

configurer rapidement des fonctions Lambda, des API, des bases de données et des sources d'événements avec seulement quelques lignes de code YAML.

AWS Cloud Development Kit (AWS CDK)

Le CDK est une approche d'IaC axée sur le code. Vous pouvez définir votre architecture basée sur Lambda à l'aide de Typescript, JavaScript, Python, Java, C#/.Net ou Go. Choisissez votre langage préféré et utilisez des éléments de programmation tels que les paramètres, les conditions, les boucles, la composition et l'héritage pour définir le résultat souhaité pour votre infrastructure. Le CDK génère ensuite les modèles CloudFormation sous-jacents pour le déploiement. Pour obtenir un exemple d'utilisation de Lambda avec CDK, consultez Déploiement de fonctions Lambda avec AWS CDK.



AWS fournit également un service appelé AWS Infrastructure Composer pour développer des modèles IaC à l'aide d'une interface graphique simple. Avec Infrastructure Composer, vous concevez une architecture d'application en faisant glisser, en regroupant et en connectant les Services AWS dans un canevas visuel. Infrastructure Composer crée ensuite un modèle AWS SAM ou un modèle CloudFormation à partir de votre conception que vous pouvez utiliser pour déployer votre application.

Dans la section the section called "Utilisation d'AWS SAM et d'Infrastructure Composer" ci-dessous, Infrastructure Composer vous permet de développer un modèle pour une application sans serveur basée sur une fonction Lambda existante.

Utilisation des fonctions Lambda dans AWS SAM et Infrastructure Composer

Dans ce tutoriel, vous pouvez commencer à utiliser l'IaC avec Lambda en créant un modèle AWS SAM à partir d'une fonction Lambda existante, puis en développant une application sans serveur dans Infrastructure Composer en ajoutant d'autres ressources AWS.

Au cours de ce didacticiel, vous apprendrez certains concepts fondamentaux, tels que la manière dont les ressources AWS sont spécifiées dans AWS SAM. Vous découvrirez également comment utiliser Infrastructure Composer pour créer une application sans serveur que vous pouvez déployer à l'aide d'AWS SAM ou d'CloudFormation.

Pour compléter ce didacticiel, effectuez les tâches suivantes :

- Créer un exemple de fonction Lambda
- Utiliser la console Lambda pour afficher le modèle AWS SAM de la fonction
- Exportez la configuration de votre fonction vers une application sans serveur simple AWS Infrastructure Composer et concevez une application sans serveur basée sur la configuration de votre fonction
- Enregistrez un modèle AWS SAM mis à jour que vous pouvez utiliser comme base pour déployer votre application sans serveur

Prérequis

Dans ce tutoriel, vous allez utiliser la fonctionnalité de synchronisation locale d'Infrastructure Composer pour enregistrer votre modèle et vos fichiers de code sur votre machine de génération locale. Pour utiliser cette fonctionnalité, vous devez disposer d'un navigateur compatible avec l'API d'accès au système de fichiers, qui permet aux applications Web de lire, d'écrire et d'enregistrer des fichiers dans votre système de fichiers local. Nous vous recommandons d'utiliser Google Chrome ou Microsoft Edge. Pour plus d'informations sur l'API d'accès au système de fichiers, voir Qu'est-ce que l'API d'accès au système de fichiers ?

Création d'une fonction Lambda

Dans cette première étape, vous allez créer une fonction Lambda que vous pouvez utiliser pour terminer le reste du didacticiel. Pour simplifier les choses, vous utilisez la console Lambda pour créer une fonction de base « Hello world » à l'aide du moteur d'exécution Python 3.11.

Pour créer une fonction Lambda « Hello world » à l'aide de la console

1. Ouvrez la console Lambda.
2. Choisissez Créer une fonction.
3. Ne désélectionnez pas l'option Créer à partir de zéro puis, sous Informations de base, saisissez **`LambdaIaCDemo`** pour le nom de la fonction.
4. Pour l'environnement d'exécution, sélectionnez Python 3.11.
5. Choisissez Créer une fonction.

Afficher le modèle AWS SAM correspondant à votre fonction

Avant d'exporter la configuration de votre fonction vers Infrastructure Composer, utilisez la console Lambda pour afficher la configuration actuelle de votre fonction sous forme de modèle AWS SAM. En suivant les étapes décrites dans cette section, vous découvrirez l'anatomie d'un modèle AWS SAM et comment définir des ressources telles que les fonctions Lambda pour commencer à spécifier une application sans serveur.

Pour afficher le modèle AWS SAM correspondant à votre fonction

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez la fonction que vous venez de créer (`LambdaIaCDemo`).
3. Dans le volet de Présentation de la fonction, choisissez Modèle.

   À la place du schéma représentant la configuration de votre fonction, vous verrez un modèle AWS SAM pour votre fonction. Le modèle doit ressembler à ce qui suit.

```
# This AWS SAM template has been generated from your function's
# configuration. If your function has one or more triggers, note
# that the AWS resources associated with these triggers aren't fully
# specified in this template and include placeholder values.Open this template
# in AWS Application Composer or your favorite IDE and modify
# it to specify a serverless application with other AWS resources.
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Specification template describing your function.
Resources:
  LambdaIaCDemo:
    Type: AWS::Serverless::Function
    Properties:
```

```
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 3
      Handler: lambda_function.lambda_handler
      Runtime: python3.11
      Architectures:
        - x86_64
      EventInvokeConfig:
        MaximumEventAgeInSeconds: 21600
        MaximumRetryAttempts: 2
      EphemeralStorage:
        Size: 512
      RuntimeManagementConfig:
        UpdateRuntimeOn: Auto
      SnapStart:
        ApplyOn: None
      PackageType: Zip
      Policies:
        Statement:
          - Effect: Allow
            Action:
              - logs:CreateLogGroup
            Resource: arn:aws:logs:us-east-1:123456789012:*
          - Effect: Allow
            Action:
              - logs:CreateLogStream
              - logs:PutLogEvents
            Resource:
              - >-
                arn:aws:logs:us-east-1:123456789012:log-group:/aws/lambda/
LambdaIaCDemo:*
```

Examinons le modèle YAML pour votre fonction pour comprendre certains concepts clés.

Le modèle commence par la déclaration `Transform: AWS::Serverless-2016-10-31`. Cette déclaration est obligatoire car, en arrière-plan, les modèles AWS SAM sont déployés via CloudFormation. L'utilisation de l'instruction `Transform` identifie le modèle en tant que fichier modèle AWS SAM.

Après la déclaration `Transform` vient la section `Resources`. C'est ici que sont définies les ressources AWS que vous souhaitez déployer avec votre modèle AWS SAM. Les modèles AWS

SAM peuvent contenir une combinaison de ressources AWS SAM et de ressources CloudFormation. Cela est dû au fait que pendant le déploiement, les modèles AWS SAM se transforment en modèles CloudFormation, de sorte que toute CloudFormation syntaxe valide peut être ajoutée à un modèle AWS SAM.

Pour le moment, une seule ressource est définie dans la section `Resources` du modèle, votre fonction Lambda `LambdaIaCDemo`. Pour ajouter une fonction Lambda à un modèle AWS SAM, vous devez utiliser le type de ressource `AWS::Serverless::Function`. La ressource `Properties` d'une fonction Lambda définit l'exécution de la fonction, le gestionnaire de fonctions et les autres options de configuration. Le chemin d'accès au code source de votre fonction qui doit être utilisé par AWS SAM pour déployer la fonction est également défini ici. Pour en savoir plus sur les ressources de fonction Lambda dans AWS SAM, consultez AWS::Serverless::Function dans le Guide du développeur AWS SAM.

Outre les propriétés et les configurations des fonctions, le modèle spécifie également une politique Gestion des identités et des accès AWS (IAM) pour votre fonction. Cette stratégie autorise votre fonction à écrire des journaux dans Amazon CloudWatch Logs. Lorsque vous créez une fonction dans la console Lambda, Lambda associe automatiquement cette politique à votre fonction. Pour en savoir plus sur la spécification d'une politique IAM pour une fonction dans un modèle AWS SAM, consultez la propriété `policies` à la page AWS::Serverless::Function du Guide du développeur AWS SAM.

Pour en savoir plus sur la structure des modèles AWS SAM, consultez la section Anatomie des modèles AWS SAM.

Utiliser AWS Infrastructure Composer pour concevoir une application sans serveur

Pour commencer à créer une application sans serveur simple en utilisant le modèle AWS SAM de votre fonction comme point de départ, vous devez exporter la configuration de votre fonction vers Infrastructure Composer et activer le mode de synchronisation local d'Infrastructure Composer. La synchronisation locale enregistre automatiquement le code de votre fonction et votre modèle AWS SAM sur votre machine de génération locale et synchronise votre modèle enregistré lorsque vous ajoutez d'autres ressources AWS dans Infrastructure Composer.

Pour exporter votre fonction vers Infrastructure Composer

1. Dans le volet Vue d'ensemble des fonctions, choisissez Exporter vers Application Composer.

   Pour exporter la configuration et le code de votre fonction vers Infrastructure Composer, Lambda crée un compartiment Amazon S3 dans votre compte pour stocker temporairement ces données.

2. Dans la boîte de dialogue, choisissez Confirmer et créer un projet pour accepter le nom par défaut de ce compartiment et exporter la configuration et le code de votre fonction vers Infrastructure Composer.
3. (Facultatif) Pour choisir un autre nom pour le compartiment Amazon S3 créé par Lambda, entrez un nouveau nom et choisissez Confirmer et créer un projet. Les noms de compartiment Amazon S3 doivent être uniques et respecter les règles de dénomination de compartiment.

   Sélectionnez Confirmer et créer un projet pour ouvrir la console Infrastructure Composer. Sur le canevas, votre fonction Lambda apparaît.
4. Dans le menu déroulant, choisissez Activer la synchronisation locale.
5. Dans la boîte de dialogue qui apparaît, choisissez Sélectionner un dossier et un dossier sur votre machine de construction locale.
6. Choisissez Activer pour activer la synchronisation locale.

Pour exporter votre fonction vers Infrastructure Composer, vous avez besoin d'une autorisation pour utiliser certaines actions d'API. Si vous ne parvenez pas à exporter votre fonction, vérifiez the section called "Autorisations requises" et assurez-vous que vous disposez des autorisations nécessaires.

> **Note**
>
> La tarification standard d'Amazon S3 s'applique au compartiment créé par Lambda lorsque vous exportez une fonction vers Infrastructure Composer. Les objets que Lambda place dans le compartiment sont automatiquement supprimés au bout de 10 jours, mais Lambda ne supprime pas le compartiment lui-même.
> Pour éviter que des frais supplémentaires ne soient ajoutés à votre Compte AWS, suivez les instructions de la section Supprimer un compartiment après avoir exporté votre fonction vers Infrastructure Composer. Pour en savoir plus sur le compartiment Amazon S3 créé par Lambda, consultez the section called "Compositeur d'infrastructure".

Pour concevoir votre application sans serveur dans Infrastructure Composer

Après avoir activé la synchronisation locale, les modifications que vous apportez dans Infrastructure Composer seront reflétées dans le modèle AWS SAM enregistré sur votre machine de génération locale. Vous pouvez désormais glisser-déposer des ressources AWS supplémentaires sur le canevas d'Infrastructure Composer pour créer votre application. Dans cet exemple, vous ajoutez

une file d'attente simple Amazon SQS comme déclencheur pour votre fonction Lambda et une table DynamoDB pour la fonction dans laquelle écrire des données.

1. Ajoutez un déclencheur Amazon SQS à votre fonction Lambda en procédant comme suit :

   a. Dans le champ de recherche de la palette Ressources, entrez **`SQS`**.

   b. Faites glisser la ressource SQS Queue sur votre canevas et positionnez-la à gauche de votre fonction Lambda.

   c. Choisissez Détails, puis, pour Logical ID, entrez **`LambdaIaCQueue`**.

   d. Choisissez Enregistrer.

   e. Connectez vos ressources Amazon SQS et Lambda en cliquant sur le port Abonnement sur la carte de file d'attente SQS et en le faisant glisser vers le port gauche de la carte de fonction Lambda. L'apparition d'une ligne entre les deux ressources indique que la connexion a abouti. Infrastructure Composer affiche également un message au bas du canevas indiquant que les deux ressources sont correctement connectées.

2. Ajoutez une table Amazon DynamoDB dans laquelle votre fonction Lambda pourra y écrire des données en procédant comme suit :

   a. Dans le champ de recherche de la palette Ressources, entrez **`DynamoDB`**.

   b. Faites glisser la ressource Table DynamoDB sur votre canevas et positionnez-la à droite de votre fonction Lambda.

   c. Choisissez Détails, puis, pour Logical ID, entrez **`LambdaIaCTable`**.

   d. Choisissez Enregistrer.

   e. Connectez la table DynamoDB à votre fonction Lambda en cliquant sur le port droit de la carte de fonction Lambda et en la faisant glisser vers le port gauche de la carte DynamoDB.

Maintenant que vous avez ajouté ces ressources supplémentaires, examinons le modèle AWS SAM mis à jour qu'Infrastructure Composer a créé.

Pour afficher votre modèle AWS SAM mis à jour

- Sur le canevas Infrastructure Composer, choisissez Modèle pour passer de la vue canevas à la vue modèle.

Votre modèle AWS SAM doit désormais contenir les ressources et propriétés supplémentaires suivantes :

- File d'attente Amazon SQS avec l'identifiant `LambdaIaCQueue`

```
LambdaIaCQueue:
    Type: AWS::SQS::Queue
    Properties:
      MessageRetentionPeriod: 345600
```

Lorsque vous ajoutez une file d'attente Amazon SQS à l'aide d'Infrastructure Composer, Infrastructure Composer définit la propriété `MessageRetentionPeriod`. Vous pouvez également définir la propriété `FifoQueue` en sélectionnant Détails sur la carte SQS Queue et en cochant ou décochant la file d'attente Fifo.

Pour définir d'autres propriétés pour votre file d'attente, vous pouvez modifier manuellement le modèle et les y ajouter. Pour en savoir plus sur la ressource `AWS::SQS::Queue` et ses propriétés disponibles, consultez AWS::SQS::Queue dans le Guide de l'utilisateur CloudFormation.

- Propriété `Events` de votre définition de fonction Lambda qui spécifie la file d'attente Amazon SQS comme déclencheur de la fonction

```
Events:
  LambdaIaCQueue:
    Type: SQS
    Properties:
      Queue: !GetAtt LambdaIaCQueue.Arn
      BatchSize: 1
```

La propriété `Events` est constituée d'un type d'événement et d'un ensemble de propriétés qui dépendent du type. Pour en savoir plus sur les différents Services AWS que vous pouvez configurer pour déclencher une fonction Lambda et sur les propriétés que vous pouvez définir, consultez EventSource dans le Guide du développeur AWS SAM.

- Table DynamoDB avec l'identifiant `LambdaIaCTable`

```
LambdaIaCTable:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        - AttributeName: id
          AttributeType: S
      BillingMode: PAY_PER_REQUEST
      KeySchema:
```

```
        - AttributeName: id
          KeyType: HASH
      StreamSpecification:
        StreamViewType: NEW_AND_OLD_IMAGES
```

Lorsque vous ajoutez une table DynamoDB à l'aide d'Infrastructure Composer, vous pouvez définir les clés de votre table en choisissant Détails sur la fiche de table DynamoDB et en modifiant les valeurs clés. Infrastructure Composer définit également des valeurs par défaut pour un certain nombre d'autres propriétés, notamment `BillingMode` et `StreamViewType`.

Pour en savoir plus sur ces propriétés et les autres propriétés que vous pouvez ajouter à votre modèle AWS SAM, consultez AWS::DynamoDB::Table dans le Guide de l'utilisateur CloudFormation.

- Une nouvelle politique IAM qui autorise votre fonction à effectuer des opérations CRUD sur la table DynamoDB que vous avez ajoutée.

```
Policies:
...
  - DynamoDBCrudPolicy:
    TableName: !Ref LambdaIaCTable
```

Le modèle AWS SAM final complet doit ressembler à ce qui suit.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Specification template describing your function.
Resources:
  LambdaIaCDemo:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 3
      Handler: lambda_function.lambda_handler
      Runtime: python3.11
      Architectures:
        - x86_64
      EventInvokeConfig:
        MaximumEventAgeInSeconds: 21600
```

```
        MaximumRetryAttempts: 2
      EphemeralStorage:
        Size: 512
      RuntimeManagementConfig:
        UpdateRuntimeOn: Auto
      SnapStart:
        ApplyOn: None
      PackageType: Zip
      Policies:
        - Statement:
            - Effect: Allow
              Action:
                - logs:CreateLogGroup
              Resource: arn:aws:logs:us-east-1:594035263019:*
            - Effect: Allow
              Action:
                - logs:CreateLogStream
                - logs:PutLogEvents
              Resource:
                - arn:aws:logs:us-east-1:594035263019:log-group:/aws/lambda/
LambdaIaCDemo:*
        - DynamoDBCrudPolicy:
            TableName: !Ref LambdaIaCTable
      Events:
        LambdaIaCQueue:
          Type: SQS
          Properties:
            Queue: !GetAtt LambdaIaCQueue.Arn
            BatchSize: 1
      Environment:
        Variables:
          LAMBDAIACTABLE_TABLE_NAME: !Ref LambdaIaCTable
          LAMBDAIACTABLE_TABLE_ARN: !GetAtt LambdaIaCTable.Arn
  LambdaIaCQueue:
    Type: AWS::SQS::Queue
    Properties:
      MessageRetentionPeriod: 345600
  LambdaIaCTable:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        - AttributeName: id
          AttributeType: S
      BillingMode: PAY_PER_REQUEST
```

```
      KeySchema:
        - AttributeName: id
          KeyType: HASH
      StreamSpecification:
        StreamViewType: NEW_AND_OLD_IMAGES
```

Déployer votre application sans serveur en utilisant AWS SAM (facultatif)

Si vous souhaitez utiliser AWS SAM pour déployer une application sans serveur à l'aide du modèle que vous venez de créer dans Infrastructure Composer, vous devez d'abord installer l'CLI d'AWS SAM. Pour ce faire, suivez les instructions de la section Installation de la CLI AWS SAM.

Avant de déployer votre application, vous devez également mettre à jour le code de fonction enregistré par Infrastructure Composer avec votre modèle. Pour le moment, le fichier `lambda_function.py` enregistré par Infrastructure Composer ne contient que le code de base « Hello world » fourni par Lambda lors de la création de la fonction.

Pour mettre à jour votre code de fonction, copiez le code suivant et collez-le dans le fichier `lambda_function.py` qu'Infrastructure Composer a enregistré sur votre machine de compilation locale. Vous avez spécifié le répertoire dans lequel Infrastructure Composer doit enregistrer ce fichier lorsque vous avez activé le mode de synchronisation locale.

Ce code accepte une paire clé-valeur dans un message provenant de la file d'attente Amazon SQS que vous avez créée dans Infrastructure Composer. Si la clé et la valeur sont des chaînes, le code les utilise ensuite pour écrire un élément dans la table DynamoDB définie dans votre modèle.

Mise à jour du code de la fonction Python

```
import boto3
import os
import json

# define the DynamoDB table that Lambda will connect to
tablename = os.environ['LAMBDAIACTABLE_TABLE_NAME']

# create the DynamoDB resource
dynamo = boto3.client('dynamodb')

def lambda_handler(event, context):
    # get the message out of the SQS event
    message = event['Records'][0]['body']
```

```
    data = json.loads(message)
    # write event data to DDB table
    if check_message_format(data):
        key = next(iter(data))
        value = data[key]
        dynamo.put_item(
            TableName=tablename,
            Item={
                'id': {'S': key},
                'Value': {'S': value}
            }
        )
    else:
        raise ValueError("Input data not in the correct format")

# check that the event object contains a single key value
# pair that can be written to the database
def check_message_format(message):
    if len(message) != 1:
        return False

    key, value = next(iter(message.items()))

    if not (isinstance(key, str) and isinstance(value, str)):
        return False

    else:
        return True
```

Pour déployer votre application sans serveur

Pour déployer votre application à l'aide d'AWS SAM CLI, procédez comme suit. Pour que votre fonction soit correctement construite et déployée, la version 3.11 de Python doit être installée sur votre machine de compilation et sur votre PATH.

1. Exécutez la commande suivante depuis le répertoire dans lequel Infrastructure Composer a enregistré vos fichiers `template.yaml` et `lambda_function.py`.

   ```
   sam build
   ```

   Cette commande rassemble les artefacts de compilation pour votre application et les place dans le format et l'emplacement appropriés pour les déployer.

2. Pour déployer votre application et créer les ressources Lambda, Amazon SQS et DynamoDB spécifiées dans votre modèle AWS SAM, exécutez la commande suivante.

   ```
   sam deploy --guided
   ```

   L'utilisation du drapeau `--guided` signifie qu'AWS SAM affiche des invites pour vous guider tout au long du processus de déploiement. Pour ce déploiement, acceptez les options par défaut en appuyant sur Entrée.

Au cours du processus de déploiement, AWS SAM crée les ressources suivantes dans votre Compte AWS :

- Une pile CloudFormation nommée `sam-app`
- Une fonction Lambda avec le format de nom `sam-app-LambdaIaCDemo-`*`99VXPpYQVv1M`*
- Une file d'attente Amazon SQS au format de nom `sam-app-LambdaIaCQueue-`*`xL87VeKsGiIo`*
- Une table DynamoDB au format de nom `sam-app-LambdaIaCTable-`*`CN0S66C0VLNV`*

AWS SAM crée également les rôles et politiques IAM nécessaires pour que votre fonction Lambda puisse lire les messages de la file d'attente Amazon SQS et effectuer des opérations CRUD sur la table DynamoDB.

Testing de votre application déployée (facultatif)

Pour vérifier que votre application sans serveur s'est déployée correctement, envoyez un message à votre file d'attente Amazon SQS contenant une paire clé-valeur et vérifiez que Lambda écrit un élément dans votre table DynamoDB en utilisant ces valeurs.

Tester votre application sans serveur

1. Ouvrez la page Files d'attente de la console Amazon SQS et sélectionnez la file d'attente créée par AWS SAM depuis votre modèle. Le nom est au format `sam-app-LambdaIaCQueue-`*`xL87VeKsGiIo`*.
2. Choisissez Envoyer et recevoir des messages, puis collez le code JSON suivant dans le champ Corps du message, dans la section Envoyer un message.

   ```
   {
       "myKey": "myValue"
   ```

```
}
```

3. Choisissez Send Message (Envoyer un message).

   L'envoi de votre message à la file d'attente amène Lambda à invoquer votre fonction par le biais du mappage de la source d'événement défini dans votre modèle AWS SAM. Pour confirmer que Lambda a invoqué votre fonction comme prévu, indiquez qu'un élément a été ajouté à votre table DynamoDB.
4. Ouvrez la page Tables de la console DynamoDB et choisissez votre table. Le nom est au format `sam-app-LambdaIaCTable-CN0S66C0VLNV`.
5. Choisissez Explorer les éléments de la table. Dans le volet Items returned (Éléments retournés), vous devriez voir un élément avec l'id `myKey` et la Valeur `myValue`.

Déploiement de fonctions Lambda avec AWS CDK

AWS Cloud Development Kit (AWS CDK) Il s'agit d'un framework d'infrastructure en tant que code (IaC) que vous pouvez utiliser pour définir une infrastructure AWS cloud en utilisant le langage de programmation de votre choix. Pour définir votre propre infrastructure de cloud, écrivez d'abord une application (dans l'un des langages pris en charge par le CDK) contenant une ou plusieurs piles. Ensuite, vous le synthétisez dans un CloudFormation modèle et déployez vos ressources sur votre Compte AWS. Suivez les étapes décrites dans cette rubrique pour déployer une fonction Lambda qui renvoie un événement depuis un point de terminaison Amazon API Gateway.

La bibliothèque AWS Construct, incluse dans le CDK, fournit des modules que vous pouvez utiliser pour modéliser les ressources Services AWS fournies. Pour les services populaires, la bibliothèque fournit des constructions organisées avec des valeurs par défaut intelligentes et les bonnes pratiques. Vous pouvez utiliser le module aws_lambda pour définir votre fonction et les ressources de support en quelques lignes de code.

Conditions préalables

Avant de commencer ce didacticiel, installez le AWS CDK en exécutant la commande suivante.

```
npm install -g aws-cdk
```

Étape 1 : Configurez votre AWS CDK projet

Créez un répertoire pour votre nouvelle AWS CDK application et initialisez le projet.

JavaScript

```
mkdir hello-lambda
cd hello-lambda
cdk init --language javascript
```

TypeScript

```
mkdir hello-lambda
cd hello-lambda
cdk init --language typescript
```

Python

```
mkdir hello-lambda
cd hello-lambda
cdk init --language python
```

Une fois le projet démarré, activez l'environnement virtuel du projet et installez les dépendances de base d' AWS CDK.

```
source .venv/bin/activate
python -m pip install -r requirements.txt
```

Java

```
mkdir hello-lambda
cd hello-lambda
cdk init --language java
```

Importez ce projet Maven dans votre environnement de développement intégré (IDE) Java. Par exemple, dans Eclipse, utilisez Fichier, Importer, Maven, Projets Maven existants.

C#

```
mkdir hello-lambda
cd hello-lambda
cdk init --language csharp
```

> **Note**
>
> Le modèle AWS CDK d'application utilise le nom du répertoire du projet pour générer des noms pour les classes et les fichiers sources. Dans cet exemple, le répertoire est nommé `hello-lambda`. Si vous utilisez un nom répertoire différent, votre application ne correspondra pas à ces instructions.

AWS CDK v2 inclut des constructions stables pour tous Services AWS dans un seul package appelé`aws-cdk-lib`. Ce package est installé en tant que dépendance lorsque vous initialisez le projet. Lorsque vous travaillez avec certains langages de programmation, le package est installé lorsque vous créez le projet pour la première fois.

Étape 2 : définir la AWS CDK pile

Une pile CDK est un ensemble d'une ou plusieurs constructions qui définissent AWS les ressources. Chaque pile CDK représente une CloudFormation pile dans votre application CDK.

Pour définir votre pile CDK, suivez les instructions de votre langage de programmation préféré. Cette pile définit les éléments suivants :

- Le nom logique de la fonction : `MyFunction`
- L'emplacement du code de fonction, spécifié dans la propriété `code`. Pour plus d'informations, consultez Handler code dans la Référence de l'API AWS Cloud Development Kit (AWS CDK) .
- Le nom logique de l'API REST : `HelloApi`
- Le nom logique du point de terminaison d'API Gateway : `ApiGwEndpoint`

Notez que toutes les piles CDK de ce tutoriel utilisent l'environnement d'exécution Node.js pour la fonction Lambda. Vous pouvez utiliser différents langages de programmation pour la pile CDK et la fonction Lambda afin de tirer parti des points forts de chaque langage. Par exemple, vous pouvez utiliser la pile CDK TypeScript pour tirer parti des avantages du typage statique pour votre code d'infrastructure. Vous pouvez utiliser la fonction Lambda JavaScript pour tirer parti de la flexibilité et du développement rapide d'un langage typé dynamiquement.

JavaScript

Ouvrez le fichier `lib/hello-lambda-stack.js` et remplacez le contenu par ce qui suit.

```
const { Stack } = require('aws-cdk-lib');
const lambda = require('aws-cdk-lib/aws-lambda');
const apigw = require('aws-cdk-lib/aws-apigateway');

class HelloLambdaStack extends Stack {
  /**
   *
   * @param {Construct} scope
   * @param {string} id
   * @param {StackProps=} props
   */
  constructor(scope, id, props) {
    super(scope, id, props);
    const fn = new lambda.Function(this, 'MyFunction', {
      code: lambda.Code.fromAsset('lib/lambda-handler'),
      runtime: lambda.Runtime.NODEJS_LATEST,
      handler: 'index.handler'
    });

    const endpoint = new apigw.LambdaRestApi(this, 'MyEndpoint', {
      handler: fn,
      restApiName: "HelloApi"
    });

  }
}

module.exports = { HelloLambdaStack }
```

TypeScript

Ouvrez le fichier `lib/hello-lambda-stack.ts` et remplacez le contenu par ce qui suit.

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as apigw from "aws-cdk-lib/aws-apigateway";
import * as lambda from "aws-cdk-lib/aws-lambda";
import * as path from 'node:path';

export class HelloLambdaStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps){
    super(scope, id, props)
    const fn = new lambda.Function(this, 'MyFunction', {
```

```
      runtime: lambda.Runtime.NODEJS_LATEST,
      handler: 'index.handler',
      code: lambda.Code.fromAsset(path.join(__dirname, 'lambda-handler')),
    });

    const endpoint = new apigw.LambdaRestApi(this, `ApiGwEndpoint`, {
      handler: fn,
      restApiName: `HelloApi`,
    });

  }
}
```

Python

Ouvrez le fichier `/hello-lambda/hello_lambda/hello_lambda_stack.py` et remplacez le contenu par ce qui suit.

```
from aws_cdk import (
    Stack,
    aws_apigateway as apigw,
    aws_lambda as _lambda
)
from constructs import Construct

class HelloLambdaStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        fn = _lambda.Function(
            self,
            "MyFunction",
            runtime=_lambda.Runtime.NODEJS_LATEST,
            handler="index.handler",
            code=_lambda.Code.from_asset("lib/lambda-handler")
        )

        endpoint = apigw.LambdaRestApi(
            self,
            "ApiGwEndpoint",
            handler=fn,
            rest_api_name="HelloApi"
```

```
        )
```

Java

Ouvrez le fichier `/hello-lambda/src/main/java/com/myorg/HelloLambdaStack.java` et remplacez le contenu par ce qui suit.

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.apigateway.LambdaRestApi;
import software.amazon.awscdk.services.lambda.Function;

public class HelloLambdaStack extends Stack {
    public HelloLambdaStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public HelloLambdaStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        Function hello = Function.Builder.create(this, "MyFunction")

 .runtime(software.amazon.awscdk.services.lambda.Runtime.NODEJS_LATEST)

 .code(software.amazon.awscdk.services.lambda.Code.fromAsset("lib/lambda-handler"))
                            .handler("index.handler")
                            .build();

        LambdaRestApi api = LambdaRestApi.Builder.create(this, "ApiGwEndpoint")
                                .restApiName("HelloApi")
                                .handler(hello)
                                .build();

    }
}
```

C#

Ouvrez le fichier `src/HelloLambda/HelloLambdaStack.cs` et remplacez le contenu par ce qui suit.

```
using Amazon.CDK;
using Amazon.CDK.AWS.APIGateway;
using Amazon.CDK.AWS.Lambda;
using Constructs;

namespace HelloLambda
{
    public class HelloLambdaStack : Stack
    {
        internal HelloLambdaStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            var fn = new Function(this, "MyFunction", new FunctionProps
            {
                Runtime = Runtime.NODEJS_LATEST,
                Code = Code.FromAsset("lib/lambda-handler"),
                Handler = "index.handler"
            });

            var api = new LambdaRestApi(this, "ApiGwEndpoint", new
 LambdaRestApiProps
            {
                Handler = fn
            });
        }
    }
}
```

Étape 3 : créer le code de la fonction Lambda

1. À partir de la racine de votre projet (`hello-lambda`), créez le répertoire `/lib/lambda-handler` pour le code de la fonction Lambda. Ce répertoire est spécifié dans la `code` propriété de votre AWS CDK pile.
2. Dans le répertoire `/lib/lambda-handler`, créez un nouveau fichier appelé `index.js`. Collez le code suivant dans le fichier. La fonction extrait des propriétés spécifiques de la requête d'API et les renvoie sous forme de réponse JSON.

```
exports.handler = async (event) => {
  // Extract specific properties from the event object
  const { resource, path, httpMethod, headers, queryStringParameters, body } =
 event;
  const response = {
    resource,
    path,
    httpMethod,
    headers,
    queryStringParameters,
    body,
  };
  return {
    body: JSON.stringify(response, null, 2),
    statusCode: 200,
  };
};
```

Étape 4 : Déployer la AWS CDK pile

1. À la racine de votre projet, effectuez la commande cdk synth :

```
cdk synth
```

Cette commande synthétise un AWS CloudFormation modèle à partir de votre pile de CDK. Le modèle est un fichier YAML d'environ 400 lignes, semblable à ce qui suit.

> **Note**
>
> Si vous obtenez l'erreur suivante, assurez-vous que vous êtes à la racine du répertoire de votre projet.
>
> ```
> --app is required either in command-line, in cdk.json or in ~/.cdk.json
> ```

Example CloudFormation modèle

```
Resources:
```

```
 MyFunctionServiceRole3C357FF2:
   Type: AWS::IAM::Role
   Properties:
     AssumeRolePolicyDocument:
       Statement:
         - Action: sts:AssumeRole
           Effect: Allow
           Principal:
             Service: lambda.amazonaws.com
       Version: "2012-10-17"
     ManagedPolicyArns:
       - Fn::Join:
           - ""
           - - "arn:"
             - Ref: AWS::Partition
             - :iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
   Metadata:
     aws:cdk:path: HelloLambdaStack/MyFunction/ServiceRole/Resource
 MyFunction1BAA52E7:
   Type: AWS::Lambda::Function
   Properties:
     Code:
       S3Bucket:
         Fn::Sub: cdk-hnb659fds-assets-${AWS::AccountId}-${AWS::Region}
       S3Key:
ab1111111cd32708dc4b83e81a21c296d607ff2cdef00f1d7f48338782f92l3901.zip
     Handler: index.handler
     Role:
       Fn::GetAtt:
         - MyFunctionServiceRole3C357FF2
         - Arn
     Runtime: nodejs24.x
     ...
```

2. Exécutez la commande cdk deploy :

```
cdk deploy
```

Patientez que vos ressources soient créées. Le résultat final inclut l'URL de votre point de terminaison d'API Gateway. Exemple :

```
Outputs:
```

```
HelloLambdaStack.ApiGwEndpoint77F417B1 = https://abcd1234.execute-api.us-
east-1.amazonaws.com/prod/
```

Étape 5 : tester de la fonction

Pour invoquer la fonction Lambda, copiez le point de terminaison d'API Gateway et collez-le dans un navigateur Web ou exécutez une commande `curl` :

```
curl -s https://abcd1234.execute-api.us-east-1.amazonaws.com/prod/
```

La réponse est une représentation JSON des propriétés sélectionnées à partir de l'objet d'événement d'origine, qui contient des informations sur la requête envoyée au point de terminaison d'API Gateway. Exemple :

```
{
  "resource": "/",
  "path": "/",
  "httpMethod": "GET",
  "headers": {
    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/
webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7",
    "Accept-Encoding": "gzip, deflate, br, zstd",
    "Accept-Language": "en-US,en;q=0.9",
    "CloudFront-Forwarded-Proto": "https",
    "CloudFront-Is-Desktop-Viewer": "true",
    "CloudFront-Is-Mobile-Viewer": "false",
    "CloudFront-Is-SmartTV-Viewer": "false",
    "CloudFront-Is-Tablet-Viewer": "false",
    "CloudFront-Viewer-ASN": "16509",
    "CloudFront-Viewer-Country": "US",
    "Host": "abcd1234.execute-api.us-east-1.amazonaws.com",
     ...
```

Étape 6 : Nettoyer vos ressources

Le point de terminaison d'API Gateway est accessible au public. Pour éviter des frais inattendus, exécutez la commande cdk destroy pour supprimer la pile et toutes les ressources associées.

```
cdk destroy
```

Étapes suivantes

Pour plus d'informations sur l'écriture d' AWS CDK applications dans la langue de votre choix, consultez les rubriques suivantes :

TypeScript

- Travailler avec le AWS CDK in TypeScript

JavaScript

- Travailler avec le AWS CDK in JavaScript

Python

- Travailler avec le AWS CDK en Python

Java

- Travailler avec le AWS CDK en Java

C#

- Travailler avec le AWS CDK en C#

Go

- Travailler avec le AWS CDK in Go

Powertools pour AWS Lambda

Powertools pour AWS Lambda (également appelé Powertools pour AWS) fournit des fonctions utilitaires, des décorateurs et des intergiciels qui gèrent les tâches Lambda courantes, comme la journalisation structurée, le suivi, la collecte de métriques et la validation des entrées. Utilisez Powertools pour AWS Lambda pour mettre en œuvre les meilleures pratiques sans serveur et accélérer le développement à travers plusieurs fonctions Lambda. Vous simplifierez ainsi les tâches de développement courantes dans vos fonctions Lambda.

Principaux avantages de Powertools pour AWS

Bien que le développement Lambda soit possible sans Powertools pour AWS, son utilisation présente plusieurs avantages :

- Observabilité intégrée : journalisation structurée, suivi et métriques personnalisées
- Gestion des secrets : récupération des paramètres, gestion des secrets et idempotence
- Amélioration progressive : choisissez les utilitaires qui répondent le mieux à vos besoins
- Développement accéléré : analyse des événements, validation et traitement par lots
- Bonnes pratiques : mise en œuvre de modèles AWS Well-Architected sans serveur

Intégration de Powertools à AWS

Powertools pour AWS vous aide à créer des applications sans serveur prêtes pour la production avec moins de code personnalisé. Disponible en Python, TypeScript/Node.js, .NET et Java, Powertools pour AWS peut être inclus via des couches Lambda ou à l'aide du gestionnaire de packages de langage. Chaque implémentation de langage fournit des fonctionnalités de base, comme la journalisation structurée, le suivi, la collecte de métriques et la gestion des événements, tout en conservant les expressions idiomatiques propres à chaque langage de programmation. Ces implémentations sont complétées par des composants spécialisés pour l'intégration des services AWS, la prise en charge de la récupération des paramètres, le traitement par lots et la gestion des API, ainsi que des bonnes pratiques, comme la propagation des ID de corrélation, la gestion des erreurs et les modèles d'idempotence. Ensemble, ces fonctionnalités permettent aux développeurs de créer des applications sans serveur robustes et faciles à maintenir tout en réduisant la charge liée au code personnalisé.

- Powertools pour AWS Lambda (Python)
- Powertools pour AWS Lambda (TypeScript)
- Powertools pour AWS Lambda (Java)
- Powertools pour AWS Lambda (.NET)

Étapes suivantes

Pour en savoir plus sur l'utilisation de Powertools pour AWS, consultez les ressources suivantes :

- Atelier Powertools pour AWS Lambda
- Modèles sans serveur qui utilisent Powertools pour AWS
- Cadre sans serveur AWS Well-Architected
- Création d'API sans serveur avec Powertools pour AWS Lambda

Gestion des flux de travail et des événements Lambda

Lorsque vous créez des applications sans serveur avec Lambda, vous avez souvent besoin de moyens pour orchestrer l'exécution des fonctions et gérer les événements. AWS propose plusieurs approches pour coordonner les fonctions Lambda :

- Fonctions durables Lambda pour une orchestration de flux de travail axée sur le code au sein de Lambda
- AWS Step Functions pour une orchestration visuelle du flux de travail entre plusieurs services
- Amazon EventBridge Scheduler et Amazon EventBridge pour les architectures et la planification axées sur les événements

Vous pouvez également intégrer ces approches ensemble. Par exemple, vous pouvez utiliser EventBridge Scheduler pour déclencher des fonctions durables ou des flux de travail Step Functions lorsque des événements spécifiques se produisent, ou configurer des flux de travail pour publier des événements sur EventBridge Scheduler à des points d'exécution définis. Les rubriques suivantes de cette section fournissent des informations supplémentaires sur la manière dont vous pouvez utiliser ces options d'orchestration.

Orchestration axée sur le code avec fonctions durables

Les fonctions Lambda durables offrent une approche d'orchestration des flux de travail axée sur le code, vous permettant de créer des flux de travail dynamiques et de longue durée directement dans vos fonctions Lambda. Contrairement aux services d'orchestration externes, les fonctions durables maintiennent la logique de votre flux de travail dans le code, ce qui facilite la mise à jour, le test et la maintenance parallèlement à votre logique métier.

Les fonctions durables sont idéales lorsque vous avez besoin de :

- Définition du flux de travail axée sur le code : définissez les flux de travail à l'aide de langages de programmation familiers plutôt que de JSON ou de concepteurs visuels
- Processus de longue durée : exécutez des flux de travail pouvant s'exécuter jusqu'à un an, bien au-delà de la limite de 15 minutes prévue par les fonctions Lambda standard
- Développement simplifié : conservez la logique du flux de travail et la logique métier dans la même base de code pour faciliter la maintenance et les tests
- Temps d'attente rentable : interrompez l'exécution pendant les états d'attente sans consommer de ressources informatiques

- Gestion d'état intégrée : point de contrôle automatique et persistance de l'état sans configuration de stockage externe

Choisir entre les fonctions durables et les Step Functions

Les fonctions durables et les Step Functions fournissent toutes deux des capacités d'orchestration des flux de travail, mais elles répondent à différents cas d'utilisation :

Considération	Fonctions durables	Step Functions
Définition du flux de travail	Le code d'abord (JavaScriptPython, Java)	Amazon States Language ou concepteur visuel basé sur JSON
Approche de développement	Base de code unique avec logique métier	Définition du flux de travail et code de fonction distincts
Intégration de service	Grâce au code de fonction Lambda et AWS SDKs	Intégrations natives avec de nombreux services AWS
Durée d'exécution	Jusqu'à 1 an	Jusqu'à 1 an (Standard), 5 minutes (Express)
Traitement parallèle	Promise.all () et modèles basés sur le code	État parallèle et carte distribuée
Gestion des erreurs	Blocs Try-catch et logique de nouvelle tentative personnalisée	États de nouvelle tentative et de capture intégrés
Surveillance visuelle	CloudWatch journaux et tableaux de bord personnalisés	Graphique d'exécution visuel et historique détaillé
Idéal pour	Flux de travail centrés sur le développeur, logique métier complexe, prototypage rapide	Orchestration multiservice, flux de travail visuels, gouvernance d'entreprise

Utilisez des fonctions durables lorsque :

- Votre équipe préfère les approches de développement axées sur le code
- La logique du flux de travail est étroitement liée à la logique métier
- Vous avez besoin d'un prototypage et d'une itération rapides
- Vos flux de travail impliquent principalement des fonctions Lambda et de simples appels de service

Utilisez Step Functions lorsque :

- Vous avez besoin d'une conception et d'un suivi visuels du flux de travail
- Vos flux de travail orchestrent de manière approfondie plusieurs AWS services
- Vous avez besoin de fonctionnalités de gouvernance et de conformité d'entreprise
- Les parties prenantes non techniques doivent comprendre la logique du flux de travail

Pour plus d'informations sur les fonctions durables, consultez la section Fonctions durables pour Lambda.

Orchestration de flux de travail avec Step Functions

AWS Step Functions est un service d'orchestration de flux de travail qui vous aide à coordonner plusieurs fonctions Lambda et AWS d'autres services dans des flux de travail structurés. Ces flux de travail peuvent maintenir l'état, gérer les erreurs grâce à des mécanismes de nouvelle tentative sophistiqués et traiter les données à grande échelle.

Step Functions propose deux types de flux de travail pour répondre aux différents besoins d'orchestration :

Flux de travail standard

> Idéal pour les flux de travail auditables de longue durée qui nécessitent une sémantique d'exécution unique. Les flux de travail standard peuvent s'exécuter pendant un an au maximum, fournissent un historique d'exécution détaillé et prennent en charge le débogage visuel. Ils conviennent aux processus comme l'exécution des commandes, les pipelines de traitement des données ou les tâches d'analytique en plusieurs étapes.

Flux de travail express

Conçu pour les charges high-event-rate de travail de courte durée avec sémantique at-least-once d'exécution. Les flux de travail express peuvent s'exécuter pendant jusqu'à cinq minutes et sont idéaux pour le traitement d'événements volumineux, les transformations de données de diffusion en continu ou les scénarios d'ingestion de données IoT. Ils offrent un débit plus élevé et un coût potentiellement inférieur par rapport aux flux de travail standard.

> **Note**
>
> Pour plus d'informations sur les types de flux de travail Step Functions, consultez Choix du type de flux de travail dans Step Functions.

Dans le cadre de ces flux de travail, Step Functions fournit deux types d'états Carte pour le traitement parallèle :

Inline Map

Traite les éléments d'un tableau JSON dans l'historique d'exécution du flux de travail parent. Inline Map prend en charge jusqu'à 40 itérations simultanées et convient aux petits jeux de données ou lorsque vous devez conserver tous les traitements dans une seule exécution. Pour plus d'informations, consultez Utilisation de l'état Map en mode Inline.

Plan distribué

Permet le traitement de charges de travail parallèles à grande échelle en itérant sur des jeux de données supérieurs à 256 Kio ou nécessitant plus de 40 itérations simultanées. Prenant en charge jusqu'à 10 000 exécutions parallèles de flux de travail enfants, Distributed Map excelle dans le traitement des données semi-structurées stockées dans Amazon S3, comme les fichiers JSON ou CSV, ce qui le rend idéal pour le traitement par lots et les opérations ETL. Pour plus d'informations, consultez Utilisation de l'état Map en mode Distributed.

En combinant ces types de flux de travail et les états Map, Step Functions fournit un ensemble d'outils flexible et puissant pour orchestrer des applications sans serveur complexes, qu'il s'agisse d'opérations à petite échelle ou de pipelines de traitement de données à grande échelle.

Pour commencer à utiliser Lambda avec Step Functions, consultez Orchestration de fonctions Lambda avec Step Functions.

Gestion des événements avec EventBridge et EventBridge planificateur

Amazon EventBridge est un service de bus d'événements qui vous aide à créer des architectures axées sur les événements. Il achemine les événements entre les AWS services, les applications intégrées et les applications SaaS (Software as a Service). EventBridge Le planificateur est un planificateur sans serveur qui vous permet de créer, d'exécuter et de gérer des tâches à partir d'un service central, en vous permettant d'invoquer des fonctions Lambda selon un calendrier à l'aide d'expressions cron et rate, ou de configurer des appels ponctuels.

Amazon EventBridge et EventBridge Scheduler vous aident à créer des architectures axées sur les événements avec Lambda. EventBridge achemine les événements entre les AWS services, les applications intégrées et les applications SaaS, tandis que EventBridge Scheduler fournit des fonctionnalités de planification spécifiques pour appeler les fonctions Lambda de manière récurrente ou ponctuelle.

Ces services fournissent plusieurs fonctionnalités clés pour travailler avec les fonctions Lambda :

- Créez des règles qui font correspondre les événements aux fonctions Lambda et les acheminent vers ces fonctions à l'aide de EventBridge
- Configurez des invocations de fonctions récurrentes à l'aide d'expressions cron et rate avec Scheduler EventBridge
- Configurer des invocations de fonction ponctuelles à des dates et heures spécifiques
- Définissez des fenêtres horaires flexibles et des politiques de nouvelle tentative pour les invocations planifiées

Pour de plus amples informations, veuillez consulter Invocation d’une fonction Lambda dans une planification.

Fonctions durables Lambda

Les fonctions durables de Lambda vous permettent de créer des applications résilientes en plusieurs étapes et des flux de travail d'IA pouvant être exécutés pendant un an tout en maintenant une progression fiable malgré les interruptions. Lorsqu'une fonction durable est exécutée, ce cycle de vie complet est appelé exécution durable, qui utilise des points de contrôle pour suivre la progression et rétablir automatiquement les défaillances par le biais d'une rediffusion, en la réexécutant depuis le début tout en omettant le travail terminé.

Dans chaque fonction, vous utilisez des opérations durables comme éléments fondamentaux. Les étapes exécutent la logique métier grâce aux nouvelles tentatives intégrées et au suivi de la progression, tandis que les temps d'attente suspendent l'exécution sans frais de calcul, ce qui les rend idéales pour les processus de longue durée tels que les human-in-the-loop flux de travail ou l'analyse des dépendances externes. Qu'il s'agisse de traiter des commandes, de coordonner des microservices ou d'orchestrer des applications d'IA agentic, les fonctions durables maintiennent automatiquement l'état et se rétablissent en cas de panne pendant que vous écrivez du code dans des langages de programmation familiers.

Principaux avantages

Écrivez du code résilient naturellement : avec des structures de programmation familières, vous pouvez écrire du code qui gère automatiquement les défaillances. Grâce au point de contrôle intégré, aux nouvelles tentatives transparentes et à la restauration automatique, votre logique métier reste claire et ciblée.

Payez uniquement pour ce que vous utilisez : pendant les opérations d'attente, votre fonction est suspendue sans frais de calcul. Pour les flux de travail de longue durée qui attendent des heures ou des jours, vous ne payez que pour le temps de traitement réel, et non pour les temps d'attente inactifs.

Simplicité opérationnelle : avec le modèle sans serveur de Lambda, vous bénéficiez d'une mise à l'échelle automatique scale-to-zero, y compris sans gérer l'infrastructure. Les fonctions durables gèrent automatiquement la gestion des états, la logique des nouvelles tentatives et la reprise en cas de panne, réduisant ainsi les frais opérationnels.

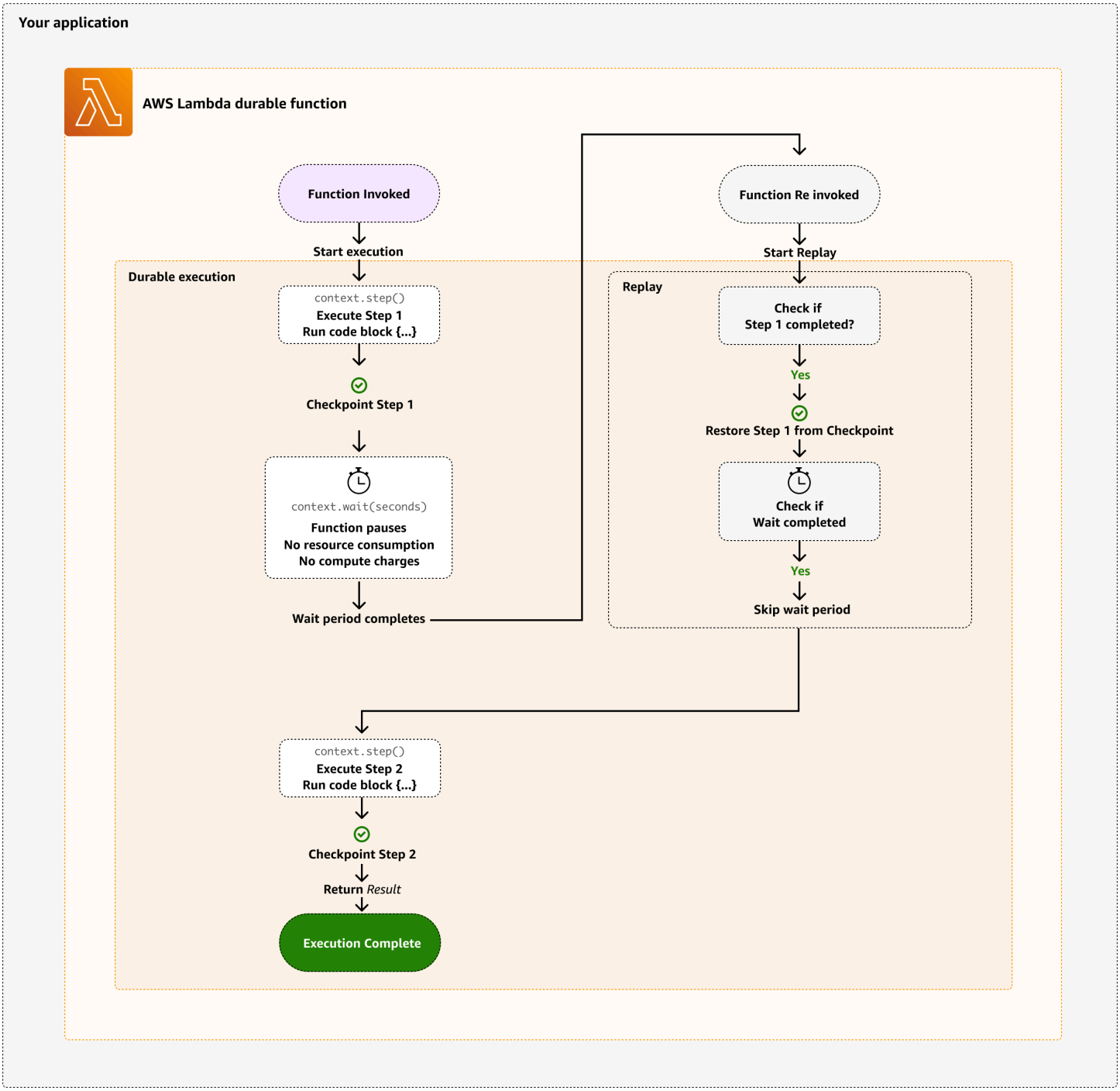
Comment ça marche

En résumé, les fonctions durables sont des fonctions Lambda classiques utilisant un checkpoint/replay mécanisme permettant de suivre les progrès et de soutenir les opérations de longue durée grâce à des points de suspension définis par l'utilisateur, communément appelés « exécution durable ». Lorsqu'une fonction durable reprend après un point d'attente ou une interruption, comme de nouvelles tentatives, le système effectue une rediffusion. Pendant la rediffusion, votre code s'exécute depuis le début mais ignore les points de contrôle terminés, utilisant les résultats enregistrés au lieu de réexécuter les opérations terminées. Ce mécanisme de rediffusion garantit la cohérence tout en permettant des exécutions de longue durée.

Une fois que votre fonction a repris après une pause ou une interruption, le système effectue une rediffusion. Pendant la rediffusion, votre code s'exécute depuis le début mais ignore les points de contrôle terminés, utilisant les résultats enregistrés au lieu de réexécuter les opérations terminées. Ce mécanisme de rediffusion garantit la cohérence tout en permettant des exécutions de longue durée.

Pour exploiter ce checkpoint-and-replay mécanisme dans vos applications, Lambda fournit un SDK d'exécution durable. Le SDK élimine la complexité de la gestion des points de contrôle et de la rediffusion en exposant des primitives simples appelées opérations durables que vous utilisez dans votre code. Le SDK est disponible pour JavaScript Python et s'intègre parfaitement à votre flux de travail de développement Lambda existant. TypeScript

Avec le SDK, vous pouvez encapsuler votre gestionnaire d'événements Lambda, qui fournit ensuite DurableContext un lien avec votre événement. Ce contexte vous donne accès à des opérations durables telles que les étapes et les attentes. Vous écrivez la logique de votre fonction sous forme de code séquentiel normal, mais au lieu d'appeler directement les services, vous regroupez ces appels par étapes pour des points de contrôle et des tentatives automatiques. Lorsque vous devez suspendre l'exécution, vous ajoutez des temps d'attente qui suspendent votre fonction sans encourir de frais. Le SDK gère toute la complexité de la gestion des états et de la rediffusion en arrière-plan, afin que votre code reste propre et lisible.



Quand utiliser des fonctions durables

Coordination de courte durée : coordonnez les paiements, les stocks et les expéditions entre plusieurs services avec annulation automatique en cas de panne. Traitez les commandes par le biais de la validation, de l'autorisation de paiement, de l'allocation des stocks et de l'exécution avec une garantie d'exécution.

Traitez les paiements en toute confiance : créez des flux de paiement résilients qui maintiennent l'état des transactions en cas d'échec et gèrent automatiquement les nouvelles tentatives. Coordonnez l'autorisation en plusieurs étapes, les contrôles antifraude et le règlement entre les fournisseurs de paiement avec une auditabilité totale à chaque étape.

Créez des flux de travail d'IA fiables : créez des flux de travail d'IA en plusieurs étapes qui enchaînent les appels, intègrent les commentaires humains et gèrent les tâches de longue durée de manière déterministe en cas de panne. Reprenez automatiquement après la suspension et ne payez que pour le temps d'exécution actif.

Orchestrez le traitement des commandes complexes : coordonnez le traitement des commandes entre les systèmes d'inventaire, de paiement, d'expédition et de notification grâce à une résilience intégrée. Gérez automatiquement les défaillances partielles, préservez l'état des commandes malgré les interruptions et attendez efficacement les événements externes sans consommer de ressources informatiques.

Automatisez les flux de travail commerciaux en plusieurs étapes : créez des flux de travail fiables pour l'intégration des employés, l'approbation des prêts et les processus de conformité qui s'étendent sur des jours ou des semaines. Maintenez l'état du flux de travail lors des approbations humaines, des intégrations de systèmes et des tâches planifiées tout en offrant une visibilité complète sur l'état et l'historique des processus.

Étapes suivantes

- Commencez avec des fonctions durables
- Découvrez le SDK d'exécution durable
- Surveillez et déboguez les fonctions durables
- Vérifiez la sécurité et les autorisations
- Suivez les meilleures pratiques

Concepts de base

Lambda fournit une exécution durable SDKs pour JavaScript TypeScript, et Python. Ils SDKs constituent la base de la création de fonctions durables, fournissant les primitives dont vous avez besoin pour contrôler la progression, gérer les nouvelles tentatives et gérer le flux d'exécution. Pour obtenir une documentation complète et des exemples du SDK, consultez le TypeScript SDKJavaScript/et le SDK Python sur. GitHub

Exécution durable

Une exécution durable représente le cycle de vie complet d'une fonction durable Lambda, utilisant un point de contrôle et un mécanisme de réexécution pour suivre l'évolution de la logique métier, suspendre l'exécution et rétablir les opérations en cas de défaillance. Lorsque les fonctions reprennent après une suspension ou une interruption, les points de contrôle précédemment complétés sont rejoués et la fonction poursuit son exécution.

Le cycle de vie peut inclure plusieurs invocations d'une fonction Lambda pour terminer l'exécution, en particulier après des suspensions ou une reprise après défaillance. Cette approche permet à votre fonction de fonctionner pendant de longues périodes (jusqu'à un an) tout en maintenant une progression fiable malgré les interruptions.

Comment fonctionne le replay

Lambda tient un journal de toutes les opérations durables (étapes, attentes et autres opérations) au fur et à mesure de l'exécution de votre fonction. Lorsque votre fonction doit être mise en pause ou rencontre une interruption, Lambda enregistre ce journal de points de contrôle et arrête l'exécution. Au moment de reprendre, Lambda invoque à nouveau votre fonction depuis le début et rejoue le journal des points de contrôle, en substituant des valeurs stockées aux opérations terminées. Cela signifie que votre code s'exécute à nouveau, mais que les étapes précédemment effectuées ne sont pas réexécutées. Leurs résultats enregistrés sont utilisés à la place.

Ce mécanisme de rediffusion est fondamental pour comprendre les fonctions durables. Votre code doit être déterministe lors de la rediffusion, c'est-à-dire qu'il produit les mêmes résultats avec les mêmes entrées. Évitez les opérations comportant des effets secondaires (comme la génération de nombres aléatoires ou l'obtention de l'heure actuelle) en dehors des étapes, car elles peuvent produire des valeurs différentes lors de la rediffusion et entraîner un comportement non déterministe.

DurableContext

DurableContextest l'objet de contexte que reçoit votre fonction durable. Il fournit des méthodes pour des opérations durables, telles que des étapes et des attentes qui créent des points de contrôle et gèrent le flux d'exécution.

Votre fonction durable reçoit un contexte Lambda `DurableContext` au lieu du contexte par défaut :

TypeScript

```
import {
```

```
  DurableContext,
  withDurableExecution,
} from "@aws/durable-execution-sdk-js";

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const result = await context.step(async () => {
      return "step completed";
    });
    return result;
  },
);
```

Python

```
from aws_durable_execution_sdk_python import (
    DurableContext,
    durable_execution,
    durable_step,
)

@durable_step
def my_step(step_context, data):
    # Your business logic
    return result

@durable_execution
def handler(event, context: DurableContext):
    result = context.step(my_step(event["data"]))
    return result
```

Le SDK Python pour les fonctions durables utilise des méthodes synchrones et ne les prend pas en charge. `await` Le TypeScript SDK utilise`async/await`.

Étapes

Steps exécute une logique métier avec des tentatives intégrées et des points de contrôle automatiques. Chaque étape enregistre son résultat, ce qui garantit que votre fonction peut reprendre à partir de n'importe quelle étape terminée après une interruption.

TypeScript

```
// Each step is automatically checkpointed
const order = await context.step(async () => processOrder(event));
const payment = await context.step(async () => processPayment(order));
const result = await context.step(async () => completeOrder(payment));
```

Python

```
# Each step is automatically checkpointed
order = context.step(lambda: process_order(event))
payment = context.step(lambda: process_payment(order))
result = context.step(lambda: complete_order(payment))
```

États d'attente

Les états d'attente sont des pauses planifiées pendant lesquelles votre fonction cesse de fonctionner (et de se recharger) jusqu'à ce qu'il soit temps de continuer. Utilisez-les pour attendre des périodes, des rappels externes ou des conditions spécifiques.

TypeScript

```
// Wait for 1 hour without consuming resources
await context.wait({ seconds:3600 });

// Wait for external callback
const approval = await context.waitForCallback(
  async (callbackId) => sendApprovalRequest(callbackId)
);
```

Python

```
# Wait for 1 hour without consuming resources
context.wait(3600)

# Wait for external callback
approval = context.wait_for_callback(
    lambda callback_id: send_approval_request(callback_id)
)
```

Lorsque votre fonction attend ou doit faire une pause, Lambda enregistre le journal des points de contrôle et arrête l'exécution. Au moment de reprendre, Lambda invoque à nouveau votre fonction et rejoue le journal des points de contrôle, en substituant des valeurs stockées aux opérations terminées.

Pour les flux de travail plus complexes, les fonctions Lambda durables sont également associées à des opérations avancées telles que `parallel()` l'exécution simultanée, le traitement `map()` de tableaux, les opérations imbriquées et `runInChildContext()` les interrogations. `waitForCondition()` Reportez-vous à la section Exemples pour obtenir des exemples détaillés et des conseils sur le moment d'utiliser chaque opération.

Invocation d'autres fonctions

Invoke permet à une fonction durable d'appeler d'autres fonctions Lambda et d'attendre leurs résultats. La fonction appelante est suspendue pendant que la fonction invoquée s'exécute, créant ainsi un point de contrôle qui préserve le résultat. Cela vous permet de créer des flux de travail modulaires dans lesquels des fonctions spécialisées gèrent des tâches spécifiques.

`context.invoke()`À utiliser pour appeler d'autres fonctions depuis votre fonction durable. L'invocation est un point de contrôle. Ainsi, si votre fonction est interrompue une fois la fonction invoquée terminée, elle reprend avec le résultat enregistré sans avoir à réinvoquer la fonction.

TypeScript

```
// Invoke another function and wait for result
const customerData = await context.invoke(
  'validate-customer',
  'arn:aws:lambda:us-east-1:123456789012:function:customer-service:1',
  { customerId: event.customerId }
);

// Use the result in subsequent steps
const order = await context.step(async () => {
  return processOrder(customerData);
});
```

Python

```
# Invoke another function and wait for result
customer_data = context.invoke(
```

```
    'arn:aws:lambda:us-east-1:123456789012:function:customer-service:1',
    {'customerId': event['customerId']},
    name='validate-customer'
)

# Use the result in subsequent steps
order = context.step(
    lambda: process_order(customer_data),
    name='process-order'
)
```

La fonction invoquée peut être une fonction Lambda durable ou standard. Si vous invoquez une fonction durable, la fonction appelante attend la fin de l'exécution durable complète. Ce modèle est courant dans les architectures de microservices où chaque fonction gère un domaine spécifique, ce qui vous permet de composer des flux de travail complexes à partir de fonctions spécialisées et réutilisables.

> **Note**
>
> Les appels entre comptes ne sont pas pris en charge. La fonction invoquée doit se trouver dans le même AWS compte que la fonction appelante.

Configuration fonctionnelle durable

Les fonctions durables ont des paramètres de configuration spécifiques qui contrôlent le comportement d'exécution et la conservation des données. Ces paramètres sont distincts de la configuration standard de la fonction Lambda et s'appliquent à l'ensemble du cycle de vie d'exécution durable.

L'`DurableConfig`objet définit la configuration des fonctions durables :

```
{
  "ExecutionTimeout": Integer,
  "RetentionPeriodInDays": Integer
}
```

Execution timeout (Délai d'exécution)

Le délai d'exécution contrôle la durée pendant laquelle une exécution durable peut s'exécuter du début à la fin. Cela est différent du délai d'expiration de la fonction Lambda, qui contrôle la durée pendant laquelle une seule invocation de fonction peut être exécutée.

Une exécution durable peut s'étendre sur plusieurs invocations de fonctions Lambda au fur et à mesure qu'elle progresse entre les points de contrôle, les temps d'attente et les rediffusions. Le délai d'exécution s'applique au temps total écoulé pendant l'exécution durable, et non aux invocations de fonctions individuelles.

Comprendre la différence

Le délai d'expiration de la fonction Lambda (maximum 15 minutes) limite chaque appel individuel de votre fonction. Le délai d'exécution durable (maximum 1 an) limite le temps total entre le début de l'exécution et sa fin, son échec ou son expiration. Au cours de cette période, votre fonction peut être invoquée plusieurs fois pendant qu'elle traite des étapes, attend et se rétablit après un échec.

Par exemple, si vous définissez un délai d'exécution durable de 24 heures et un délai d'expiration de la fonction Lambda de 5 minutes :

- Chaque invocation de fonction doit être terminée dans les 5 minutes
- L'ensemble de l'exécution durable peut durer jusqu'à 24 heures
- Votre fonction peut être invoquée plusieurs fois pendant ces 24 heures
- Les opérations d'attente ne sont pas prises en compte dans le délai d'expiration de la fonction Lambda, mais dans le délai d'exécution

Vous pouvez configurer le délai d'exécution lors de la création d'une fonction durable à l'aide de la console Lambda AWS CLI, ou. AWS SAM Dans la console Lambda, choisissez votre fonction, puis Configuration, Exécution durable. Définissez la valeur du délai d'exécution en secondes (par défaut : 86400 secondes/24 heures, minimum : 60 secondes, maximum : 31536000 secondes/1 an).

> **Note**
>
> Le délai d'exécution et le délai d'expiration de la fonction Lambda sont des paramètres différents. La fonction Lambda timeout contrôle la durée pendant laquelle chaque appel individuel peut être exécuté (maximum 15 minutes). Le délai d'exécution contrôle le temps total écoulé pour l'ensemble de l'exécution durable (maximum 1 an).

Période de conservation

La période de conservation contrôle la durée pendant laquelle Lambda conserve l'historique des exécutions et les données des points de contrôle après la fin d'une exécution durable. Ces données incluent les résultats des étapes, l'état d'exécution et le journal complet des points de contrôle.

Une fois la période de conservation expirée, Lambda supprime l'historique d'exécution et les données des points de contrôle. Vous ne pouvez plus récupérer les détails de l'exécution ni rejouer l'exécution. La période de rétention commence lorsque l'exécution atteint un état terminal (SUCCEDED, FAILED, STOPPED ou TIMED_OUT).

Vous pouvez configurer la période de rétention lors de la création d'une fonction durable à l'aide de la console Lambda AWS CLI, ou. AWS SAM Dans la console Lambda, choisissez votre fonction, puis Configuration, Exécution durable. Définissez la valeur de la période de conservation en jours (par défaut : 14 jours, minimum : 1 jour, maximum : 90 jours).

Choisissez une période de rétention en fonction de vos exigences de conformité, de vos besoins de débogage et de considérations financières. Des périodes de rétention plus longues permettent de consacrer plus de temps au débogage et à l'audit, mais augmentent les coûts de stockage.

Création de fonctions Lambda durables

Pour commencer à utiliser les fonctions durables Lambda, utilisez la console Lambda pour créer une fonction durable. En quelques minutes, vous pouvez créer et déployer une fonction durable qui utilise des étapes et des temps d'attente pour démontrer une exécution basée sur des points de contrôle.

Au cours de ce didacticiel, vous découvrirez les concepts fondamentaux des fonctions durables, tels que la façon d'utiliser l'`DurableContext`objet, de créer des points de contrôle avec des étapes et de suspendre l'exécution avec des temps d'attente. Vous découvrirez également comment fonctionne le replay lorsque votre fonction reprend après une attente.

Pour simplifier les choses, vous devez créer votre fonction à l'aide de l'exécution Python ou Node.js. Avec ces langages interprétés, vous pouvez modifier le code de fonction directement dans l'éditeur de code intégré à la console.

> **Tip**
>
> Pour apprendre à créer des solutions sans serveur, consultez le Guide du développeur sans serveur.

Conditions préalables

Inscrivez-vous pour un Compte AWS

Si vous n'en avez pas Compte AWS, procédez comme suit pour en créer un.

Pour vous inscrire à un Compte AWS

1. Ouvrez l'https://portal.aws.amazon.com/billing/inscription.
2. Suivez les instructions en ligne.

   Dans le cadre de la procédure d'inscription, vous recevrez un appel téléphonique ou un SMS et vous saisirez un code de vérification en utilisant le clavier numérique du téléphone.

   Lorsque vous vous inscrivez à un Compte AWS, un Utilisateur racine d'un compte AWSest créé. Par défaut, seul l'utilisateur racine a accès à l'ensemble des Services AWS et des ressources de ce compte. La meilleure pratique de sécurité consiste à attribuer un accès administratif à un utilisateur, et à utiliser uniquement l'utilisateur racine pour effectuer les tâches nécessitant un accès utilisateur racine.

AWS vous envoie un e-mail de confirmation une fois le processus d'inscription terminé. À tout moment, vous pouvez consulter l'activité actuelle de votre compte et gérer votre compte en accédant à https://aws.amazon.com/et en choisissant Mon compte.

Création d'un utilisateur doté d'un accès administratif

Après vous être inscrit à un Compte AWS, sécurisez Utilisateur racine d'un compte AWS AWS IAM Identity Center, activez et créez un utilisateur administratif afin de ne pas utiliser l'utilisateur root pour les tâches quotidiennes.

Sécurisez votre Utilisateur racine d'un compte AWS

1. Connectez-vous en AWS Management Consoletant que propriétaire du compte en choisissant Utilisateur root et en saisissant votre adresse Compte AWS e-mail. Sur la page suivante, saisissez votre mot de passe.

   Pour obtenir de l'aide pour vous connecter en utilisant l'utilisateur racine, consultez Connexion en tant qu'utilisateur racine dans le Guide de l'utilisateur Connexion à AWS .
2. Activez l'authentification multifactorielle (MFA) pour votre utilisateur racine.

Pour obtenir des instructions, voir Activer un périphérique MFA virtuel pour votre utilisateur Compte AWS root (console) dans le guide de l'utilisateur IAM.

Création d’un utilisateur doté d’un accès administratif

1. Activez IAM Identity Center.

   Pour obtenir des instructions, consultez Activation d’ AWS IAM Identity Center dans le Guide de l’utilisateur AWS IAM Identity Center .
2. Dans IAM Identity Center, octroyez un accès administratif à un utilisateur.

   Pour un didacticiel sur l'utilisation du Répertoire IAM Identity Center comme source d'identité, voir Configurer l'accès utilisateur par défaut Répertoire IAM Identity Center dans le Guide de AWS IAM Identity Center l'utilisateur.

Connexion en tant qu’utilisateur doté d’un accès administratif

- Pour vous connecter avec votre utilisateur IAM Identity Center, utilisez l’URL de connexion qui a été envoyée à votre adresse e-mail lorsque vous avez créé l’utilisateur IAM Identity Center.

  Pour obtenir de l'aide pour vous connecter en utilisant un utilisateur d'IAM Identity Center, consultez la section Connexion au portail AWS d'accès dans le guide de l'Connexion à AWS utilisateur.

Attribution d’un accès à d’autres utilisateurs

1. Dans IAM Identity Center, créez un ensemble d’autorisations qui respecte la bonne pratique consistant à appliquer les autorisations de moindre privilège.

   Pour obtenir des instructions, consultez Création d’un ensemble d’autorisations dans le Guide de l’utilisateur AWS IAM Identity Center .
2. Attribuez des utilisateurs à un groupe, puis attribuez un accès par authentification unique au groupe.

   Pour obtenir des instructions, consultez Ajout de groupes dans le Guide de l’utilisateur AWS IAM Identity Center .

Créez une fonction Lambda durable avec la console

Dans cet exemple, votre fonction durable traite une commande en plusieurs étapes avec un point de contrôle automatique. La fonction prend un objet JSON contenant un numéro de commande, valide la commande, traite le paiement et confirme la commande. Chaque étape est automatiquement contrôlée. Ainsi, si la fonction est interrompue, elle reprend à partir de la dernière étape terminée.

Votre fonction illustre également une opération d'attente, interrompant l'exécution pendant une courte période pour simuler l'attente d'une confirmation externe.

Pour créer une fonction durable avec la console

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez Créer une fonction.
3. Sélectionnez Créer à partir de zéro.
4. Dans le volet Informations de base, pour Nom de la fonction, saisissez **`myDurableFunction`**.
5. Pour Runtime, choisissez Node.js 24 ou Python 3.14.
6. Sélectionnez Activer l'exécution durable.

Lambda crée votre fonction durable avec un rôle d'exécution qui inclut des autorisations pour les opérations de point de contrôle (`lambda:CheckpointDurableExecutionset`). `lambda:GetDurableExecutionState`

> **Note**
>
> Les environnements d'exécution Lambda incluent le SDK Durable Execution, qui vous permet de tester des fonctions durables sans créer de dépendances. Toutefois, nous vous recommandons d'inclure le SDK dans votre package de déploiement pour la production. Cela garantit la cohérence des versions et évite les mises à jour d'exécution potentielles susceptibles d'affecter votre fonction.

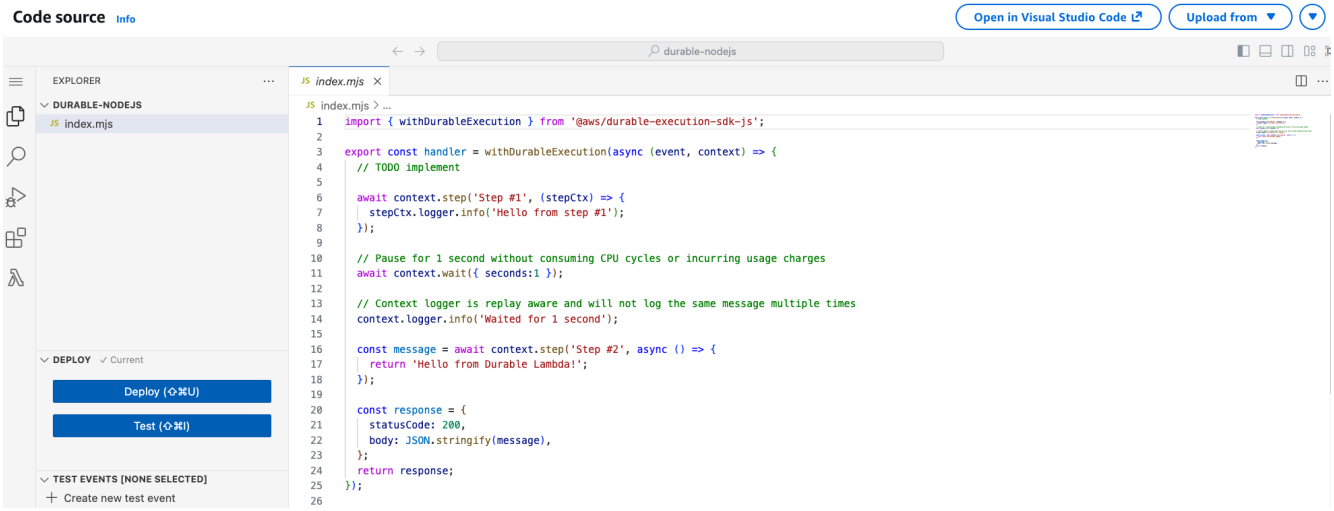
Utilisez l'éditeur de code intégré à la console pour ajouter votre code de fonction durable.

Node.js

Pour modifier le code dans la console

1. Cliquez sur l'onglet Code.

   Dans l'éditeur de code intégré de la console, vous devriez voir le code de fonction créé par Lambda. Si vous ne voyez pas l'onglet index.mjs dans l'éditeur de code, sélectionnez index.mjs dans l'explorateur de fichiers, comme le montre le schéma suivant.

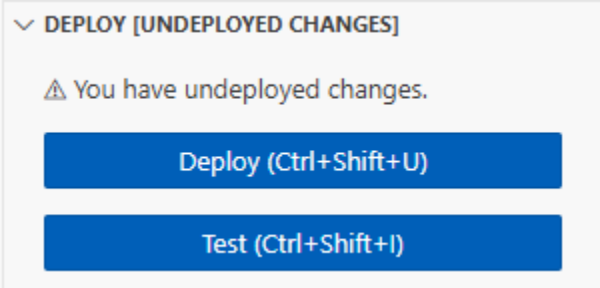


2. Collez le code suivant dans l'onglet index.mjs, en remplaçant le code créé par Lambda.

```
import {
  DurableContext,
  withDurableExecution,
} from "@aws/durable-execution-sdk-js";

export const handler = withDurableExecution(
  async (event, context) => {
    const orderId = event.orderId;

    // Step 1: Validate order
    const validationResult = await context.step(async (stepContext) => {
      stepContext.logger.info(`Validating order ${orderId}`);
      return { orderId, status: "validated" };
    });

    // Step 2: Process payment
    const paymentResult = await context.step(async (stepContext) => {
      stepContext.logger.info(`Processing payment for order ${orderId}`);
```

```
      return { orderId, status: "paid", amount: 99.99 };
    });

    // Wait for 10 seconds to simulate external confirmation
    await context.wait({ seconds: 10 });

    // Step 3: Confirm order
    const confirmationResult = await context.step(async (stepContext) => {
      stepContext.logger.info(`Confirming order ${orderId}`);
      return { orderId, status: "confirmed" };
    });

    return {
      orderId: orderId,
      status: "completed",
      steps: [validationResult, paymentResult, confirmationResult]
    };
  }
);
```

3. Dans la section DÉPLOYER, choisissez Déployer pour mettre à jour le code de votre fonction :



Comprendre votre code de fonction durable

Avant de passer à l'étape suivante, examinons le code de la fonction et comprenons les principaux concepts relatifs aux fonctions durables.

- L'`withDurableExecution`emballage :

  Votre fonction durable est emballée avec`withDurableExecution`. Ce wrapper permet une exécution durable en fournissant l'`DurableContext`objet et en gérant les opérations des points de contrôle.

- L'`DurableContext`objet :

Au lieu du contexte Lambda standard, votre fonction reçoit un. `DurableContext` Cet objet fournit des méthodes pour des opérations durables telles `wait()` que `step()` et qui créent des points de contrôle.

- Étapes et points de contrôle :

  Chaque `context.step()` appel crée un point de contrôle avant et après l'exécution. Si votre fonction est interrompue, elle reprend à partir du dernier point de contrôle terminé. La fonction ne réexécute pas les étapes terminées. Il utilise plutôt leurs résultats enregistrés.

- Opérations d'attente :

  L'`context.wait()`appel interrompt l'exécution sans consommer de ressources informatiques. Une fois l'attente terminée, Lambda appelle à nouveau votre fonction et rejoue le journal des points de contrôle, en substituant des valeurs stockées aux étapes terminées.

- Mécanisme de rediffusion :

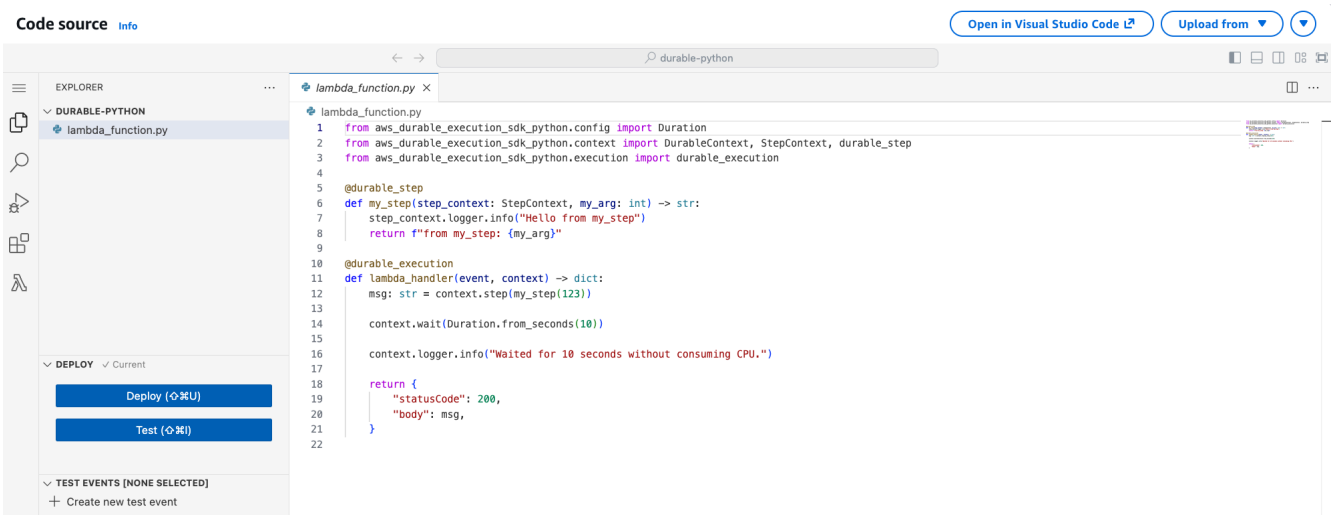
  Lorsque votre fonction reprend après une attente ou une interruption, Lambda exécute votre code depuis le début. Toutefois, les étapes terminées ne sont pas réexécutées. Lambda rejoue leurs résultats à partir du journal des points de contrôle. C'est pourquoi votre code doit être déterministe.

Python

Pour modifier le code dans la console

1. Cliquez sur l’onglet Code.

   Dans l'éditeur de code intégré de la console, vous devriez voir le code de fonction créé par Lambda. Si vous ne voyez pas l'onglet lambda_function.py dans l'éditeur de code, sélectionnez lambda_function.py dans l'explorateur de fichiers, comme le montre le schéma suivant.

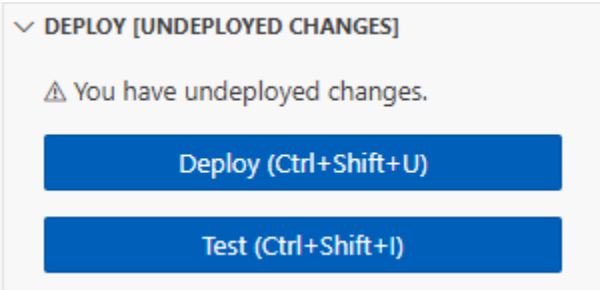


2. Collez le code suivant dans l'onglet lambda_function.py, en remplaçant le code créé par Lambda.

```
from aws_durable_execution_sdk_python import (
    DurableContext,
    durable_execution,
    durable_step,
)
from aws_durable_execution_sdk_python.config import Duration

@durable_step
def validate_order(step_context, order_id):
    step_context.logger.info(f"Validating order {order_id}")
    return {"orderId": order_id, "status": "validated"}

@durable_step
def process_payment(step_context, order_id):
    step_context.logger.info(f"Processing payment for order {order_id}")
    return {"orderId": order_id, "status": "paid", "amount": 99.99}

@durable_step
def confirm_order(step_context, order_id):
    step_context.logger.info(f"Confirming order {order_id}")
    return {"orderId": order_id, "status": "confirmed"}

@durable_execution
def lambda_handler(event, context: DurableContext):
    order_id = event['orderId']

    # Step 1: Validate order
```

```
    validation_result = context.step(validate_order(order_id))

    # Step 2: Process payment
    payment_result = context.step(process_payment(order_id))

    # Wait for 10 seconds to simulate external confirmation
    context.wait(Duration.from_seconds(10))

    # Step 3: Confirm order
    confirmation_result = context.step(confirm_order(order_id))

    return {
        "orderId": order_id,
        "status": "completed",
        "steps": [validation_result, payment_result, confirmation_result]
    }
```

3. Dans la section DÉPLOYER, choisissez Déployer pour mettre à jour le code de votre fonction :



Comprendre votre code de fonction durable

Avant de passer à l'étape suivante, examinons le code de la fonction et comprenons les principaux concepts relatifs aux fonctions durables.

- Le `@durable_execution` décorateur :

  Votre fonction de gestion est décorée avec`@durable_execution`. Ce décorateur permet une exécution durable en fournissant l'`DurableContext`objet et en gérant les opérations aux points de contrôle.
- Le `@durable_step` décorateur :

  Chaque fonction de marche est décorée avec`@durable_step`. Ce décorateur définit la fonction comme une étape durable qui crée des points de contrôle.

- L'`DurableContext`objet :

  Au lieu du contexte Lambda standard, votre fonction reçoit un. `DurableContext` Cet objet fournit des méthodes pour des opérations durables telles `wait()` que `step()` et qui créent des points de contrôle.

- Étapes et points de contrôle :

  Chaque `context.step()` appel crée un point de contrôle avant et après l'exécution. Si votre fonction est interrompue, elle reprend à partir du dernier point de contrôle terminé. La fonction ne réexécute pas les étapes terminées. Il utilise plutôt leurs résultats enregistrés.

- Opérations d'attente :

  L'`context.wait()`appel interrompt l'exécution sans consommer de ressources informatiques. Une fois l'attente terminée, Lambda appelle à nouveau votre fonction et rejoue le journal des points de contrôle, en substituant des valeurs stockées aux étapes terminées.

- Le SDK Python est synchrone :

  Notez que le SDK Python n'utilise `await` pas. Toutes les opérations durables sont des appels de méthode synchrones.

Invoquez la fonction durable à l'aide de l'éditeur de code de la console

Les fonctions durables nécessitent un ARN qualifié pour l'invocation. Avant de pouvoir invoquer votre fonction durable, publiez une version.

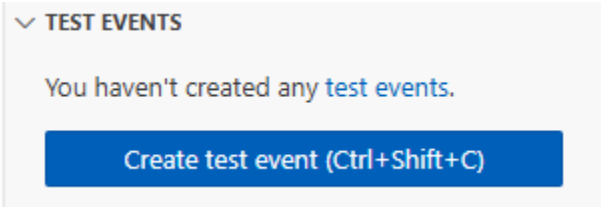
Pour publier une version de votre fonction

1. Choisissez l'onglet Versions.
2. Choisissez Publish new version (Publier une nouvelle version).
3. Dans le champ Description de la version, entrez **`Initial version`** (facultatif).
4. Choisissez Publier.
5. Lambda crée la version 1 de votre fonction. Notez que la fonction ARN inclut désormais `:1` à la fin, ce qui indique qu'il s'agit de la version 1.

Créez maintenant un événement de test à envoyer à votre fonction. L'événement est un document au format JSON contenant un numéro de commande.

Pour créer l'événement de test

1. Dans la section ÉVÉNEMENTS DE TEST de l'éditeur de code de console, choisissez Créer un événement de test.



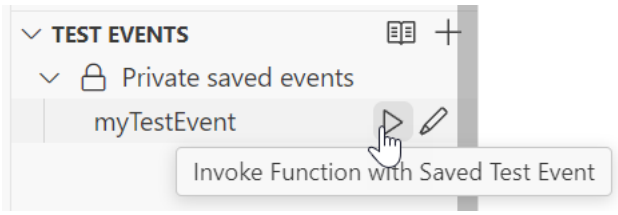
2. Dans Event Name (Nom de l'événement), saisissez **`myTestEvent`**.
3. Dans la section JSON d'événement, remplacez le JSON par défaut par ce qui suit :

```
{
  "orderId": "order-12345"
}
```

4. Choisissez Enregistrer.

Pour tester votre fonctionnalité durable et en voir l'exécution

Dans la section ÉVÉNEMENTS DE TEST de l'éditeur de code de la console, cliquez sur l'icône d'exécution à côté de votre événement de test :



Votre fonction durable commence à s'exécuter. Comme il inclut une attente de 10 secondes, l'appel initial se termine rapidement et la fonction reprend après la période d'attente. Vous pouvez consulter la progression de l'exécution dans l'onglet Exécutions durables.

Pour visualiser l'exécution durable de vos fonctions

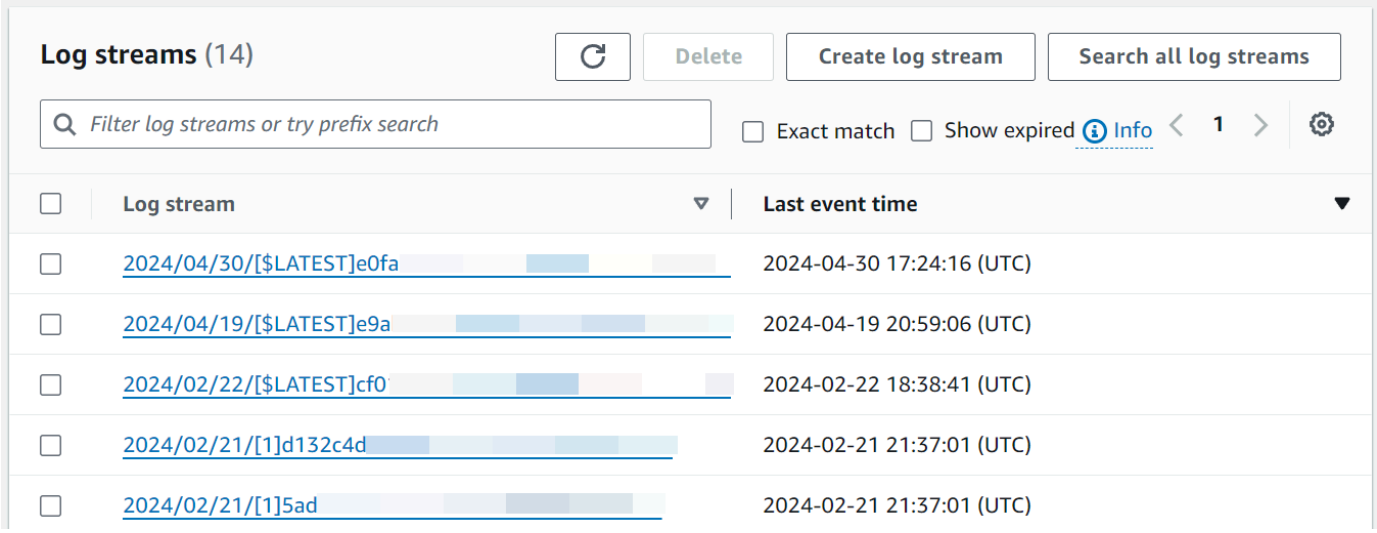
1. Choisissez l'onglet Exécutions durables.
2. Trouvez votre exécution dans la liste. L'exécution indique l'état actuel (en cours, réussi ou échec).
3. Choisissez l'ID d'exécution pour afficher les détails, notamment :

- Chronologie d'exécution indiquant la date à laquelle chaque étape est terminée
- Historique des points de contrôle
- Périodes d'attente
- Résultats des étapes

Vous pouvez également consulter les journaux de votre fonction dans CloudWatch Logs pour voir le résultat de chaque étape de la console.

Pour consulter les enregistrements d'invocation de votre fonction dans Logs CloudWatch

1. Ouvrez la page Groupes de journaux de la CloudWatch console.
2. Choisissez le groupe de journaux de votre fonction (`/aws/lambda/myDurableFunction`).
3. Faites défiler la page vers le bas et choisissez le flux de journaux pour les invocations de fonctions que vous souhaitez consulter.



   Vous devriez voir des entrées de journal pour chaque appel de votre fonction, y compris l'exécution initiale et la rediffusion après l'attente.

Nettoyage

Lorsque vous avez fini d'utiliser l'exemple de fonction durable, supprimez-le. Vous pouvez également supprimer le groupe de journaux qui stocke les journaux de la fonction, et le rôle d'exécution créé par la console.

Pour supprimer la fonction Lambda

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction que vous avez créée.
3. Sélectionnez Actions, Supprimer.
4. Saisissez **`confirm`** dans la zone de saisie de texte et choisissez Delete (Supprimer).

Pour supprimer le groupe de journaux

1. Ouvrez la page Log groups (Groupes de journaux) de la console CloudWatch.
2. Sélectionnez le groupe de journaux de la fonction (`/aws/lambda/myDurableFunction`).
3. Sélectionnez Actions, Delete log group(s) (Supprimer le ou les groupes de journaux).
4. Dans la boîte de dialogue Delete log group(s) (Supprimer le ou les groupes de journaux), sélectionnez Delete (Supprimer).

Pour supprimer le rôle d'exécution

1. Ouvrez la page Rôles de la console Gestion des identités et des accès AWS (IAM).
2. Sélectionnez le rôle d'exécution de la fonction (par exemple, `myDurableFunction-role-31exxmpl`).
3. Sélectionnez Delete (Supprimer).
4. Dans la fenêtre de dialogue Supprimer le rôle, saisissez le nom du rôle, puis sélectionnez Supprimer.

Ressources supplémentaires et prochaines étapes

Maintenant que vous avez créé et testé une fonction simple et durable à l'aide de la console, procédez comme suit :

- Découvrez les cas d'utilisation courants des fonctions durables, notamment les transactions distribuées, le traitement des commandes et les flux de travail de révision humaine. Voir les exemples.
- Découvrez comment surveiller les exécutions de fonctions durables à l'aide de CloudWatch métriques et d'un historique d'exécution. Consultez la section Surveillance et débogage.

- Découvrez comment invoquer des fonctions durables de manière synchrone et asynchrone, et comment gérer les exécutions de longue durée. Voir Invocation de fonctions durables.
- Suivez les meilleures pratiques en matière de rédaction de code déterministe, de gestion de la taille des points de contrôle et d'optimisation des coûts. Consultez la section Bonnes pratiques.
- Découvrez comment tester des fonctions durables localement et dans le cloud. Voir Tester des fonctions durables.

Déployez et invoquez des fonctions durables Lambda avec le AWS CLI

Utilisez le AWS CLI pour créer et déployer des fonctions Lambda durables avec des commandes impératives. Cette approche vous permet de contrôler directement chaque étape du processus de déploiement.

Conditions préalables

- Installez et configurez l' AWS CLI. Pour obtenir des instructions, reportez-vous à la section Installation du AWS CLI.
- Créez un package de déploiement avec votre code de fonction et le SDK d'exécution durable.
- Créez un rôle d'exécution IAM avec des autorisations de point de contrôle.

Créer le rôle d'exécution

Créez un rôle IAM avec des autorisations pour l'exécution Lambda de base et les opérations de point de contrôle.

Pour créer le rôle d'exécution

1. Créez un document de politique de confiance qui permet à Lambda d'assumer le rôle. Enregistrez ceci sous le nom `trust-policy.json` :

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
```

```
      "Action": "sts:AssumeRole"
    }
  ]
}
```

2. Créez le rôle :

```
aws iam create-role \
  --role-name durable-function-role \
  --assume-role-policy-document file://trust-policy.json
```

3. Joignez la politique d'exécution de base :

```
aws iam attach-role-policy \
  --role-name durable-function-role \
  --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
```

4. Créez une politique pour les opérations aux points de contrôle. Enregistrez ceci sous le nom `checkpoint-policy.json` :

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "lambda:CheckpointDurableExecutions",
        "lambda:GetDurableExecutionState"
      ],
      "Resource": "arn:aws:lambda:REGION:ACCOUNT_ID:function:myDurableFunction"
    }
  ]
}
```

5. Créez et joignez la politique des points de contrôle :

```
aws iam create-policy \
  --policy-name durable-checkpoint-policy \
  --policy-document file://checkpoint-policy.json

aws iam attach-role-policy \
  --role-name durable-function-role \
```

```
  --policy-arn arn:aws:iam::ACCOUNT_ID:policy/durable-checkpoint-policy
```

Créez une fonction durable

Créez votre fonction durable avec le `--durable-config` paramètre.

Pour créer une fonction durable

1. Package de votre code de fonction avec les dépendances dans un fichier .zip :

```
zip -r function.zip index.mjs node_modules/
```

2. Créez la fonction avec l'exécution durable activée :

```
aws lambda create-function \
  --function-name myDurableFunction \
  --runtime nodejs22.x \
  --role arn:aws:iam::ACCOUNT_ID:role/durable-function-role \
  --handler index.handler \
  --zip-file fileb://function.zip \
  --durable-config '{"ExecutionTimeout": 10, "RetentionPeriodInDays":1}'
```

> **Note**
>
> Vous ne pouvez activer l'exécution durable que lors de la création de la fonction. Vous ne pouvez pas l'activer sur les fonctions existantes.

Publier une version

Les fonctions durables nécessitent un ARN qualifié (avec version ou alias) pour être invoquées. Publiez une version de votre fonction :

```
aws lambda publish-version \
  --function-name myDurableFunction \
  --description "Initial version"
```

La commande renvoie l'ARN de la version. Notez le numéro de version (par exemple, `:1`) à la fin de l'ARN.

Créez éventuellement un alias pointant vers la version :

```
aws lambda create-alias \
  --function-name myDurableFunction \
  --name prod \
  --function-version 1
```

Invoquez la fonction durable

Appelez votre fonction durable à l'aide de l'ARN qualifié (version ou alias).

Invocation synchrone

Pour les exécutions qui se terminent en 15 minutes, utilisez l'invocation synchrone :

```
aws lambda invoke \
  --function-name myDurableFunction:1 \
  --payload '{"orderId": "order-12345"}' \
  --cli-binary-format raw-in-base64-out \
  response.json
```

Ou en utilisant un alias :

```
aws lambda invoke \
  --function-name myDurableFunction:prod \
  --payload '{"orderId": "order-12345"}' \
  --cli-binary-format raw-in-base64-out \
  response.json
```

invocation asynchrone

Pour les exécutions de longue durée, utilisez l'invocation asynchrone :

```
aws lambda invoke \
  --function-name myDurableFunction:prod \
  --invocation-type Event \
  --payload '{"orderId": "order-12345"}' \
  --cli-binary-format raw-in-base64-out \
  response.json
```

En cas d'invocation asynchrone, Lambda revient immédiatement. La fonction continue de s'exécuter en arrière-plan.

Gérez des exécutions durables

Utilisez les commandes suivantes pour gérer et surveiller les exécutions de fonctions durables.

Lister les exécutions

Répertoriez toutes les exécutions d'une fonction durable :

```
aws lambda list-durable-executions \
  --function-name myDurableFunction:prod
```

Obtenir les détails de l'exécution

Obtenez des informations sur une exécution spécifique :

```
aws lambda get-durable-execution \
  --function-name myDurableFunction:prod \
  --execution-id exec-abc123
```

Obtenir l'historique des exécutions

Consultez l'historique des points de contrôle d'une exécution :

```
aws lambda get-durable-execution-history \
  --function-name myDurableFunction:prod \
  --execution-id exec-abc123
```

Arrêter une exécution

Arrêtez une exécution durable en cours :

```
aws lambda stop-durable-execution \
  --function-name myDurableFunction:prod \
  --execution-id exec-abc123
```

Mettre à jour le code de la fonction

Mettez à jour votre code de fonction durable et publiez une nouvelle version :

Pour mettre à jour et publier une nouvelle version

1. Mettez à jour le code de fonction :

```
aws lambda update-function-code \
  --function-name myDurableFunction \
  --zip-file fileb://function.zip
```

2. Attendez que la mise à jour soit terminée :

```
aws lambda wait function-updated \
  --function-name myDurableFunction
```

3. Publiez une nouvelle version :

```
aws lambda publish-version \
  --function-name myDurableFunction \
  --description "Updated order processing logic"
```

4. Mettez à jour l'alias pour qu'il pointe vers la nouvelle version :

```
aws lambda update-alias \
  --function-name myDurableFunction \
  --name prod \
  --function-version 2
```

> **Important**
>
> Les exécutions en cours continuent à utiliser la version avec laquelle elles ont débuté. Les nouveaux appels utilisent la version mise à jour de l'alias.

Affichage des journaux de fonctions

Consultez les journaux de votre fonction durable dans CloudWatch Logs :

```
aws logs tail /aws/lambda/myDurableFunction --follow
```

Filtrez les journaux pour une exécution spécifique :

```
aws logs filter-log-events \
  --log-group-name /aws/lambda/myDurableFunction \
```

```
  --filter-pattern "exec-abc123"
```

nettoyer des ressources ;

Supprimez votre fonction durable et les ressources associées :

```
# Delete the function
aws lambda delete-function --function-name myDurableFunction

# Delete the IAM role policies
aws iam detach-role-policy \
  --role-name durable-function-role \
  --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole

aws iam detach-role-policy \
  --role-name durable-function-role \
  --policy-arn arn:aws:iam::ACCOUNT_ID:policy/durable-checkpoint-policy

aws iam delete-policy \
  --policy-arn arn:aws:iam::ACCOUNT_ID:policy/durable-checkpoint-policy

# Delete the role
aws iam delete-role --role-name durable-function-role
```

Étapes suivantes

Après avoir déployé votre fonction durable avec AWS CLI :

- Surveillez les exécutions à l'aide `list-durable-executions` des `get-durable-execution` commandes et
- Afficher les opérations des points de contrôle dans les événements de AWS CloudTrail données
- Configurer des CloudWatch alarmes en cas d'échec d'exécution ou d'exécutions de longue durée
- Automatisez les déploiements à l'aide de scripts shell ou CI/CD de pipelines

Pour plus d'informations sur AWS CLI les commandes pour Lambda, consultez la référence des AWS CLI commandes.

Déployez des fonctions durables Lambda avec l'infrastructure en tant que code

Vous pouvez déployer des fonctions durables Lambda à l'aide d'outils d'infrastructure en tant que code (IaC) tels que AWS CloudFormation AWS CDK, ou. AWS Serverless Application Model Ces outils vous permettent de définir votre fonction, votre rôle d'exécution et vos autorisations dans le code, ce qui rend les déploiements reproductibles et contrôlés par version.

Les trois outils vous obligent à :

- Permettre une exécution durable de la fonction
- Accorder des autorisations de point de contrôle au rôle d'exécution
- Publier une version ou créer un alias (les fonctions durables nécessitent une qualification ARNs)

AWS CloudFormation

CloudFormation À utiliser pour définir votre fonction durable dans un modèle. L'exemple suivant crée une fonction durable dotée des autorisations requises.

```
AWSTemplateFormatVersion: '2010-09-09'
Description: Lambda durable function example

Resources:
  DurableFunctionRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service: lambda.amazonaws.com
            Action: sts:AssumeRole
      ManagedPolicyArns:
        - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
      Policies:
        - PolicyName: DurableExecutionPolicy
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
```

```
              - Effect: Allow
                Action:
                  - lambda:CheckpointDurableExecutions
                  - lambda:GetDurableExecutionState
                Resource: !GetAtt DurableFunction.Arn

  DurableFunction:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: myDurableFunction
      Runtime: nodejs22.x
      Handler: index.handler
      Role: !GetAtt DurableFunctionRole.Arn
      Code:
        ZipFile: |
          // Your durable function code here
          export const handler = async (event, context) => {
            return { statusCode: 200 };
          };
      DurableConfig:
        ExecutionTimeout: 10
        RetentionPeriodInDays: 1

  DurableFunctionVersion:
    Type: AWS::Lambda::Version
    Properties:
      FunctionName: !Ref DurableFunction
      Description: Initial version

  DurableFunctionAlias:
    Type: AWS::Lambda::Alias
    Properties:
      FunctionName: !Ref DurableFunction
      FunctionVersion: !GetAtt DurableFunctionVersion.Version
      Name: prod

Outputs:
  FunctionArn:
    Description: Durable function ARN
    Value: !GetAtt DurableFunction.Arn
  AliasArn:
    Description: Function alias ARN (use this for invocations)
    Value: !Ref DurableFunctionAlias
```

Pour déployer le modèle

```
aws cloudformation deploy \
  --template-file template.yaml \
  --stack-name my-durable-function-stack \
  --capabilities CAPABILITY_IAM
```

AWS CDK

AWS CDK vous permet de définir l'infrastructure à l'aide de langages de programmation. Les exemples suivants montrent comment créer une fonction durable à l'aide TypeScript de Python.

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import * as lambda from 'aws-cdk-lib/aws-lambda';
import * as iam from 'aws-cdk-lib/aws-iam';
import { Construct } from 'constructs';

export class DurableFunctionStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Create the durable function
    const durableFunction = new lambda.Function(this, 'DurableFunction', {
      runtime: lambda.Runtime.NODEJS_22_X,
      handler: 'index.handler',
      code: lambda.Code.fromAsset('lambda'),
      functionName: 'myDurableFunction',
      durableConfig: { executionTimeout: Duration.hours(1), retentionPeriod:
 Duration.days(30) },
    });

    // Add checkpoint permissions
    durableFunction.addToRolePolicy(new iam.PolicyStatement({
      actions: [
        'lambda:CheckpointDurableExecutions',
        'lambda:GetDurableExecutionState',
      ],
      resources: [durableFunction.functionArn],
    }));

    // Create version and alias
```

```
    const version = durableFunction.currentVersion;
    const alias = new lambda.Alias(this, 'ProdAlias', {
      aliasName: 'prod',
      version: version,
    });

    // Output the alias ARN
    new cdk.CfnOutput(this, 'FunctionAliasArn', {
      value: alias.functionArn,
      description: 'Use this ARN to invoke the durable function',
    });
  }
}
```

Python

```
from aws_cdk import (
    Stack,
    aws_lambda as lambda_,
    aws_iam as iam,
    CfnOutput,
)
from constructs import Construct

class DurableFunctionStack(Stack):
    def __init__(self, scope: Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)

        # Create the durable function
        durable_function = lambda_.Function(
            self, 'DurableFunction',
            runtime=lambda_.Runtime.NODEJS_22_X,
            handler='index.handler',
            code=lambda_.Code.from_asset('lambda'),
            function_name='myDurableFunction',
            durable_execution={execution_timeout: Duration.hours(1),
 retention_period: Duration.days(30)}
        )

        # Add checkpoint permissions
        durable_function.add_to_role_policy(iam.PolicyStatement(
            actions=[
                'lambda:CheckpointDurableExecutions',
```

```
                'lambda:GetDurableExecutionState',
            ],
            resources=[durable_function.function_arn]
        ))

        # Create version and alias
        version = durable_function.current_version
        alias = lambda_.Alias(
            self, 'ProdAlias',
            alias_name='prod',
            version=version
        )

        # Output the alias ARN
        CfnOutput(
            self, 'FunctionAliasArn',
            value=alias.function_arn,
            description='Use this ARN to invoke the durable function'
        )
```

Pour déployer la pile CDK

```
cdk deploy
```

AWS Serverless Application Model

AWS SAM simplifie les CloudFormation modèles pour les applications sans serveur. Le modèle suivant crée une fonction durable avec AWS SAM.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: Lambda durable function with SAM

Resources:
  DurableFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: myDurableFunction
      Runtime: nodejs22.x
      Handler: index.handler
      CodeUri: ./src
      DurableConfig:
```

```
        ExecutionTimeout: 10
        RetentionPeriodInDays: 1
      Policies:
        - Version: '2012-10-17'
          Statement:
            - Effect: Allow
              Action:
                - lambda:CheckpointDurableExecutions
                - lambda:GetDurableExecutionState
              Resource: !Sub 'arn:aws:lambda:${AWS::Region}:${AWS::AccountId}:function:
${DurableFunction}'
      AutoPublishAlias: prod

Outputs:
  FunctionArn:
    Description: Durable function ARN
    Value: !GetAtt DurableFunction.Arn
  AliasArn:
    Description: Function alias ARN (use this for invocations)
    Value: !Ref DurableFunction.Alias
```

Pour déployer le modèle SAM

```
sam build
sam deploy --guided
```

Modèles de configuration courants

Quel que soit l'outil IaC que vous utilisez, suivez les modèles suivants pour des fonctions durables :

Permettre une exécution durable

Définition de la propriété `DurableExecution.Enabled` à `true`. Cette propriété n'est disponible que lors de la création de la fonction ; vous ne pouvez pas activer l'exécution durable sur les fonctions existantes.

Accorder des autorisations aux points de contrôle

Ajoutez `lambda:CheckpointDurableExecutions` et `lambda:GetDurableExecutionState` au rôle d'exécution. Étendez ces autorisations à l'ARN de la fonction spécifique.

Utiliser qualifié ARNs

Créez une version ou un alias pour votre fonction. Les fonctions durables nécessitent une qualification ARNs (avec version ou alias) pour être invoquées. Utilisez `AutoPublishAlias` AWS SAM ou créez des versions explicites dans CloudFormation et AWS CDK.

Dépendances de package

Incluez le SDK d'exécution durable dans votre package de déploiement. Pour Node.js, installez`@aws/durable-execution-sdk-js`. Pour Python, installez`aws-durable-execution-sdk-python`.

Étapes suivantes

Après avoir déployé votre fonction durable :

- Testez votre fonction à l'aide de l'ARN qualifié (version ou alias)
- Surveillez la progression de l'exécution dans la console Lambda sous l'onglet Exécutions durables
- Afficher les opérations des points de contrôle dans les événements de AWS CloudTrail données
- Consultez CloudWatch les journaux pour connaître le comportement des fonctions, des sorties et des rediffusions

Pour plus d'informations sur le déploiement de fonctions Lambda avec les outils IaC, voir :

- CloudFormation AWS::Lambda::Function référence
- AWS CDK Documentation du module Lambda
- AWS SAM Manuel du développeur

Exemples et cas d'utilisation

Les fonctions durables de Lambda vous permettent de créer des applications en plusieurs étapes tolérantes aux pannes en utilisant des opérations durables telles que les étapes et les attentes. Grâce au point de contrôle automatique et à un modèle de réexécution des points de contrôle, dans lequel l'exécution redémarre depuis le début après un échec mais ignore les points de contrôle terminés, vos fonctions peuvent se rétablir après un échec et reprendre l'exécution sans perdre leur progression.

Procédés de courte durée tolérants aux pannes

Utilisez des fonctions durables pour créer des opérations fiables qui se terminent généralement en quelques minutes. Bien que ces processus soient plus courts que les flux de travail de longue durée, ils bénéficient tout de même du point de contrôle automatique et de la tolérance aux pannes dans les systèmes distribués. Des fonctions durables garantissent le succès de vos processus en plusieurs étapes, même en cas d'échec d'appels de service individuels, sans nécessiter de gestion des erreurs complexe ni de code de gestion des états.

Les scénarios courants incluent les systèmes de réservation d'hôtels, les plateformes de réservation de restaurants, les demandes de voyages en covoiturage, les achats de billets pour des événements et les mises à niveau d'abonnements SaaS. Ces scénarios présentent des caractéristiques communes : plusieurs appels de service qui doivent être exécutés simultanément, la nécessité d'une nouvelle tentative automatique en cas d'échec transitoire et la nécessité de maintenir un état cohérent sur l'ensemble des systèmes distribués.

Transactions distribuées sur des microservices

Coordonnez les paiements, les stocks et les expéditions entre plusieurs services grâce à l'annulation automatique en cas de panne. Chaque opération de service est encapsulée en une étape, ce qui garantit que la transaction peut être rétablie à tout moment en cas de défaillance d'un service.

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { orderId, amount, items } = event;

    // Reserve inventory across multiple warehouses
    const inventory = await context.step("reserve-inventory", async () => {
      return await inventoryService.reserve(items);
    });

    // Process payment
    const payment = await context.step("process-payment", async () => {
      return await paymentService.charge(amount);
    });
```

```
    // Create shipment
    const shipment = await context.step("create-shipment", async () => {
      return await shippingService.createShipment(orderId, inventory);
    });

    return { orderId, status: 'completed', shipment };
  }
);
```

Python

```
from aws_durable_execution_sdk import DurableContext, durable_execution

@durable_execution
def lambda_handler(event, context: DurableContext):
    order_id = event['orderId']
    amount = event['amount']
    items = event['items']

    # Reserve inventory across multiple warehouses
    inventory = context.step(
        lambda _: inventory_service.reserve(items),
        name='reserve-inventory'
    )

    # Process payment
    payment = context.step(
        lambda _: payment_service.charge(amount),
        name='process-payment'
    )

    # Create shipment
    shipment = context.step(
        lambda _: shipping_service.create_shipment(order_id, inventory),
        name='create-shipment'
    )

    return {'orderId': order_id, 'status': 'completed', 'shipment': shipment}
```

Si une étape échoue, la fonction réessaie automatiquement à partir du dernier point de contrôle réussi. La réservation d'inventaire est maintenue même si le traitement du paiement échoue temporairement. Lorsque la fonction essaie à nouveau, elle ignore l'étape d'inventaire terminée et passe directement au traitement des paiements. Cela élimine les réservations dupliquées et garantit un état cohérent dans l'ensemble de votre système distribué.

Traitement des commandes en plusieurs étapes

Traitez les commandes par le biais de la validation, de l'autorisation de paiement, de l'allocation des stocks et de l'exécution avec une nouvelle tentative et une reprise automatiques. Chaque étape fait l'objet d'un point de contrôle, ce qui garantit que la commande progresse même si les étapes individuelles échouent et que vous réessayez.

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { orderId, customerId, items } = event;

    // Validate order details
    const validation = await context.step("validate-order", async () => {
      const customer = await customerService.validate(customerId);
      const itemsValid = await inventoryService.validateItems(items);
      return { customer, itemsValid };
    });

    if (!validation.itemsValid) {
      return { orderId, status: 'rejected', reason: 'invalid_items' };
    }

    // Authorize payment
    const authorization = await context.step("authorize-payment", async () => {
      return await paymentService.authorize(
        validation.customer.paymentMethod,
        calculateTotal(items)
      );
    });
```

```
    // Allocate inventory
    const allocation = await context.step("allocate-inventory", async () => {
      return await inventoryService.allocate(items);
    });

    // Fulfill order
    const fulfillment = await context.step("fulfill-order", async () => {
      return await fulfillmentService.createShipment({
        orderId,
        items: allocation.allocatedItems,
        address: validation.customer.shippingAddress
      });
    });

    return {
      orderId,
      status: 'completed',
      trackingNumber: fulfillment.trackingNumber
    };
  }
);
```

Python

```
from aws_durable_execution_sdk import DurableContext, durable_execution

@durable_execution
def lambda_handler(event, context: DurableContext):
    order_id = event['orderId']
    customer_id = event['customerId']
    items = event['items']

    # Validate order details
    def validate_order(_):
        customer = customer_service.validate(customer_id)
        items_valid = inventory_service.validate_items(items)
        return {'customer': customer, 'itemsValid': items_valid}

    validation = context.step(validate_order, name='validate-order')

    if not validation['itemsValid']:
```

```
        return {'orderId': order_id, 'status': 'rejected', 'reason':
'invalid_items'}

    # Authorize payment
    authorization = context.step(
        lambda _: payment_service.authorize(
            validation['customer']['paymentMethod'],
            calculate_total(items)
        ),
        name='authorize-payment'
    )

    # Allocate inventory
    allocation = context.step(
        lambda _: inventory_service.allocate(items),
        name='allocate-inventory'
    )

    # Fulfill order
    fulfillment = context.step(
        lambda _: fulfillment_service.create_shipment({
            'orderId': order_id,
            'items': allocation['allocatedItems'],
            'address': validation['customer']['shippingAddress']
        }),
        name='fulfill-order'
    )

    return {
        'orderId': order_id,
        'status': 'completed',
        'trackingNumber': fulfillment['trackingNumber']
    }
```

Ce modèle garantit que les commandes ne restent jamais bloquées dans des états intermédiaires. Si la validation échoue, la commande est rejetée avant l'autorisation de paiement. Si l'autorisation de paiement échoue, le stock n'est pas alloué. Chaque étape s'appuie sur la précédente avec une nouvelle tentative et une restauration automatiques.

Remarque

La vérification conditionnelle `if (!validation.itemsValid)` se situe en dehors d'une étape et sera réexécutée pendant la rediffusion. C'est sûr car c'est déterministe : cela produit toujours le même résultat pour le même objet de validation.

Procédés de longue durée

Utilisez des fonctions durables pour les processus qui s'étendent sur des heures, des jours ou des semaines. Les opérations d'attente suspendent l'exécution sans frais de calcul, ce qui permet de rentabiliser les processus de longue durée. Pendant les périodes d'attente, votre fonction cesse de fonctionner et Lambda recycle l'environnement d'exécution. Au moment de reprendre, Lambda invoque à nouveau votre fonction et la rejoue depuis le dernier point de contrôle.

Ce modèle d'exécution rend les fonctions durables idéales pour les processus qui doivent être interrompus pendant de longues périodes, qu'il s'agisse d'attendre des décisions humaines, des réponses du système externe, des fenêtres de traitement planifiées ou des retards temporels. Vous ne payez que pour le temps de calcul actif, et non pour le temps d'attente.

Les scénarios courants incluent les processus d'approbation des documents, le traitement par lots planifié, les processus d'intégration sur plusieurs jours, les processus d'essai d'abonnement et les systèmes de notification différée. Ces scénarios présentent des caractéristiques communes : des délais d'attente prolongés mesurés en heures ou en jours, la nécessité de maintenir l'état d'exécution pendant ces temps d'attente et des exigences liées aux coûts dans lesquelles le paiement pour le temps de calcul inactif est prohibitif.

Human-in-the-loop approbations

Suspendez l'exécution pour les révisions de documents, les approbations ou les décisions tout en maintenant l'état d'exécution. La fonction attend les rappels externes sans consommer de ressources et reprend automatiquement lorsque l'approbation est reçue.

Ce modèle est essentiel pour les processus qui nécessitent un jugement humain ou une validation externe. La fonction est suspendue au point de rappel, sans frais de calcul pendant l'attente. Lorsque quelqu'un soumet sa décision via l'API, Lambda invoque à nouveau votre fonction et rejoue depuis le point de contrôle, en continuant avec le résultat de l'approbation.

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";
```

```
export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { documentId, reviewers } = event;

    // Step 1: Prepare document for review
    const prepared = await context.step("prepare-document", async () => {
      return await documentService.prepare(documentId);
    });

    // Step 2: Request approval with callback
    const approval = await context.waitForCallback(
      "approval-callback",
      async (callbackId) => {
        await notificationService.sendApprovalRequest({
          documentId,
          reviewers,
          callbackId,
          expiresIn: 86400
        });
      },
      {
        timeout: { seconds: 86400 }
      }
    );

    // Function resumes here when approval is received
    if (approval?.approved) {
      const finalized = await context.step("finalize-document", async () => {
        return await documentService.finalize(documentId, approval.comments);
      });

      return {
        status: 'approved',
        documentId,
        finalizedAt: finalized.timestamp
      };
    }

    // Handle rejection
    await context.step("archive-rejected", async () => {
      await documentService.archive(documentId, approval?.reason);
    });
```

```
    return {
      status: 'rejected',
      documentId,
      reason: approval?.reason
    };
  }
);
```

Python

```
from aws_durable_execution_sdk import DurableContext, durable_execution, WaitConfig

@durable_execution
def lambda_handler(event, context: DurableContext):
    document_id = event['documentId']
    reviewers = event['reviewers']

    # Step 1: Prepare document for review
    prepared = context.step(
        lambda _: document_service.prepare(document_id),
        name='prepare-document'
    )

    # Step 2: Request approval with callback
    def send_approval_request(callback_id):
        notification_service.send_approval_request({
            'documentId': document_id,
            'reviewers': reviewers,
            'callbackId': callback_id,
            'expiresIn': 86400
        })

    approval = context.wait_for_callback(
        send_approval_request,
        name='approval-callback',
        config=WaitConfig(timeout=86400)
    )

    # Function resumes here when approval is received
    if approval and approval.get('approved'):
        finalized = context.step(
```

```
            lambda _: document_service.finalize(document_id,
approval.get('comments')),
            name='finalize-document'
        )

        return {
            'status': 'approved',
            'documentId': document_id,
            'finalizedAt': finalized['timestamp']
        }

    # Handle rejection
    context.step(
        lambda _: document_service.archive(document_id, approval.get('reason') if
approval else None),
        name='archive-rejected'
    )

    return {
        'status': 'rejected',
        'documentId': document_id,
        'reason': approval.get('reason') if approval else None
    }
```

Lorsque le rappel est reçu et que votre fonction reprend, elle est rediffusée depuis le début. L'étape de préparation du document renvoie instantanément le résultat du point de contrôle. L' waitForCallback opération renvoie également instantanément le résultat d'approbation enregistré au lieu d'attendre à nouveau. L'exécution se poursuit ensuite jusqu'aux étapes de finalisation ou d'archivage.

Pipelines de données en plusieurs étapes

Traitez de grands ensembles de données par le biais de phases d'extraction, de transformation et de chargement avec des points de contrôle entre les étapes. Chaque étape peut prendre des heures, et les points de contrôle garantissent que le pipeline peut reprendre à n'importe quelle étape en cas d'interruption.

Ce modèle est idéal pour les flux de travail ETL, les migrations de données ou les tâches de traitement par lots où vous devez traiter les données par étapes, avec des points de récupération entre eux. Si une étape échoue, le pipeline reprend à partir de la dernière étape terminée plutôt que

de recommencer depuis le début. Vous pouvez également utiliser les opérations d'attente pour faire une pause entre les étapes, en respectant les limites de débit, en attendant que les systèmes en aval soient prêts ou en planifiant le traitement en dehors des heures de pointe.

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { datasetId, batchSize } = event;

    // Stage 1: Extract data from source
    const extracted = await context.step("extract-data", async () => {
      const records = await sourceDatabase.extractRecords(datasetId);
      return { recordCount: records.length, records };
    });

    // Wait 5 minutes to respect source system rate limits
    await context.wait({ seconds: 300 });

    // Stage 2: Transform data in batches
    const transformed = await context.step("transform-data", async () => {
      const batches = chunkArray(extracted.records, batchSize);
      const results = [];

      for (const batch of batches) {
        const transformed = await transformService.processBatch(batch);
        results.push(transformed);
      }

      return { batchCount: batches.length, results };
    });

    // Wait until off-peak hours (e.g., 2 AM)
    const now = new Date();
    const targetHour = 2;
    const msUntilTarget = calculateMsUntilHour(now, targetHour);
    await context.wait({ seconds: Math.floor(msUntilTarget / 1000) });

    // Stage 3: Load data to destination
```

```
    const loaded = await context.step("load-data", async () => {
      let loadedCount = 0;

      for (const result of transformed.results) {
        await destinationDatabase.loadBatch(result);
        loadedCount += result.length;
      }

      return { loadedCount };
    });

    // Stage 4: Verify and finalize
    const verified = await context.step("verify-pipeline", async () => {
      const verification = await destinationDatabase.verifyRecords(datasetId);
      await pipelineService.markComplete(datasetId, verification);
      return verification;
    });

    return {
      datasetId,
      recordsProcessed: extracted.recordCount,
      batchesProcessed: transformed.batchCount,
      recordsLoaded: loaded.loadedCount,
      verified: verified.success
    };
  }
);
```

Python

```
from aws_durable_execution_sdk import DurableContext, durable_execution
from datetime import datetime

@durable_execution
def lambda_handler(event, context: DurableContext):
    dataset_id = event['datasetId']
    batch_size = event['batchSize']

    # Stage 1: Extract data from source
    def extract_data(_):
        records = source_database.extract_records(dataset_id)
```

```
        return {'recordCount': len(records), 'records': records}

    extracted = context.step(extract_data, name='extract-data')

    # Wait 5 minutes to respect source system rate limits
    context.wait(300)

    # Stage 2: Transform data in batches
    def transform_data(_):
        batches = chunk_array(extracted['records'], batch_size)
        results = []

        for batch in batches:
            transformed = transform_service.process_batch(batch)
            results.append(transformed)

        return {'batchCount': len(batches), 'results': results}

    transformed = context.step(transform_data, name='transform-data')

    # Wait until off-peak hours (e.g., 2 AM)
    now = datetime.now()
    target_hour = 2
    ms_until_target = calculate_ms_until_hour(now, target_hour)
    context.wait(ms_until_target // 1000)

    # Stage 3: Load data to destination
    def load_data(_):
        loaded_count = 0

        for result in transformed['results']:
            destination_database.load_batch(result)
            loaded_count += len(result)

        return {'loadedCount': loaded_count}

    loaded = context.step(load_data, name='load-data')

    # Stage 4: Verify and finalize
    def verify_pipeline(_):
        verification = destination_database.verify_records(dataset_id)
        pipeline_service.mark_complete(dataset_id, verification)
        return verification
```

```
    verified = context.step(verify_pipeline, name='verify-pipeline')

    return {
        'datasetId': dataset_id,
        'recordsProcessed': extracted['recordCount'],
        'batchesProcessed': transformed['batchCount'],
        'recordsLoaded': loaded['loadedCount'],
        'verified': verified['success']
    }
```

Chaque étape est encapsulée dans une étape, ce qui crée un point de contrôle qui permet au pipeline de reprendre à partir de n'importe quelle étape en cas d'interruption. L'attente de 5 minutes entre l'extraction et la transformation respecte les limites de débit du système source sans consommer de ressources informatiques, tandis que l'attente jusqu'à 2 heures du matin permet de planifier l'opération de chargement coûteuse pendant les heures creuses.

Remarque

L'`new Date()`appel et la `calculateMsUntilHour()` fonction sont des étapes extérieures et seront réexécutés pendant la rediffusion. Pour les opérations basées sur le temps qui doivent être cohérentes entre les rediffusions, calculez l'horodatage d'une étape ou utilisez-le uniquement pour les durées d'attente (qui sont des points de contrôle).

Invocations enchaînées entre les fonctions

Appelez d'autres fonctions Lambda depuis une fonction durable en utilisant. `context.invoke()` La fonction appelante est suspendue en attendant que la fonction invoquée soit terminée, créant ainsi un point de contrôle qui préserve le résultat. Si la fonction appelante est interrompue une fois la fonction invoquée terminée, elle reprend avec le résultat enregistré sans avoir à réinvoquer la fonction.

Utilisez ce modèle lorsque vous avez des fonctions spécialisées qui gèrent des domaines spécifiques (validation des clients, traitement des paiements, gestion des stocks) et que vous devez les coordonner dans un flux de travail. Chaque fonction conserve sa propre logique et peut être invoquée par plusieurs fonctions d'orchestrateur, évitant ainsi la duplication de code.

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";

// Main orchestrator function
export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { orderId, customerId } = event;

    // Step 1: Validate customer by invoking customer service function
    const customer = await context.invoke(
      "validate-customer",
      "arn:aws:lambda:us-east-1:123456789012:function:customer-service:1",
      { customerId }
    );

    if (!customer.isValid) {
      return { orderId, status: "rejected", reason: "invalid_customer" };
    }

    // Step 2: Check inventory by invoking inventory service function
    const inventory = await context.invoke(
      "check-inventory",
      "arn:aws:lambda:us-east-1:123456789012:function:inventory-service:1",
      { orderId, items: event.items }
    );

    if (!inventory.available) {
      return { orderId, status: "rejected", reason: "insufficient_inventory" };
    }

    // Step 3: Process payment by invoking payment service function
    const payment = await context.invoke(
      "process-payment",
      "arn:aws:lambda:us-east-1:123456789012:function:payment-service:1",
      {
        customerId,
        amount: inventory.totalAmount,
        paymentMethod: customer.paymentMethod
      }
    );

    // Step 4: Create shipment by invoking fulfillment service function
    const shipment = await context.invoke(
      "create-shipment",
```

```
      "arn:aws:lambda:us-east-1:123456789012:function:fulfillment-service:1",
      {
        orderId,
        items: inventory.allocatedItems,
        address: customer.shippingAddress
      }
    );

    return {
      orderId,
      status: "completed",
      trackingNumber: shipment.trackingNumber,
      estimatedDelivery: shipment.estimatedDelivery
    };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import DurableContext, durable_execution

# Main orchestrator function
@durable_execution
def lambda_handler(event, context: DurableContext):
    order_id = event['orderId']
    customer_id = event['customerId']

    # Step 1: Validate customer by invoking customer service function
    customer = context.invoke(
        'arn:aws:lambda:us-east-1:123456789012:function:customer-service:1',
        {'customerId': customer_id},
        name='validate-customer'
    )

    if not customer['isValid']:
        return {'orderId': order_id, 'status': 'rejected', 'reason':
 'invalid_customer'}

    # Step 2: Check inventory by invoking inventory service function
    inventory = context.invoke(
        'arn:aws:lambda:us-east-1:123456789012:function:inventory-service:1',
```

```
        {'orderId': order_id, 'items': event['items']},
        name='check-inventory'
    )

    if not inventory['available']:
        return {'orderId': order_id, 'status': 'rejected', 'reason':
'insufficient_inventory'}

    # Step 3: Process payment by invoking payment service function
    payment = context.invoke(
        'arn:aws:lambda:us-east-1:123456789012:function:payment-service:1',
        {
            'customerId': customer_id,
            'amount': inventory['totalAmount'],
            'paymentMethod': customer['paymentMethod']
        },
        name='process-payment'
    )

    # Step 4: Create shipment by invoking fulfillment service function
    shipment = context.invoke(
        'arn:aws:lambda:us-east-1:123456789012:function:fulfillment-service:1',
        {
            'orderId': order_id,
            'items': inventory['allocatedItems'],
            'address': customer['shippingAddress']
        },
        name='create-shipment'
    )

    return {
        'orderId': order_id,
        'status': 'completed',
        'trackingNumber': shipment['trackingNumber'],
        'estimatedDelivery': shipment['estimatedDelivery']
    }
```

Chaque invocation crée un point de contrôle dans la fonction d'orchestrateur. Si l'orchestrateur est interrompu une fois la validation du client terminée, il reprend à partir de ce point de contrôle avec les données client stockées, en ignorant l'appel de validation. Cela permet d'éviter les appels dupliqués vers les services en aval et de garantir une exécution cohérente malgré les interruptions.

Les fonctions invoquées peuvent être des fonctions Lambda durables ou standard. Si vous invoquez une fonction durable, elle peut avoir son propre flux de travail en plusieurs étapes avec des temps d'attente et des points de contrôle. L'orchestrateur attend simplement la fin de l'exécution durable complète pour obtenir le résultat final.

> **Note**
>
> Les appels entre comptes ne sont pas pris en charge. Toutes les fonctions invoquées doivent être dans le même AWS compte que la fonction appelante.

Modèles avancés

Utilisez des fonctions durables pour créer des applications complexes en plusieurs étapes qui combinent plusieurs opérations durables, l'exécution parallèle, le traitement de tableaux, la logique conditionnelle et le sondage. Ces modèles vous permettent de créer des applications sophistiquées qui coordonnent de nombreuses tâches tout en maintenant la tolérance aux pannes et la restauration automatique.

Les modèles avancés vont au-delà de simples étapes séquentielles. Vous pouvez exécuter des opérations simultanément`parallel()`, traiter des tableaux avec ces primitives`map()`, attendre les conditions extérieures et combiner ces primitives pour créer des applications fiables. `waitForCondition()` Chaque opération durable crée ses propres points de contrôle, afin que votre application puisse se rétablir à tout moment en cas d'interruption.

Processus d'intégration des utilisateurs

Guidez les utilisateurs lors de l'enregistrement, de la vérification des e-mails, de la configuration du profil et de la configuration initiale avec gestion des nouvelles tentatives. Cet exemple combine des étapes séquentielles, des rappels et une logique conditionnelle pour créer un processus d'intégration complet.

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
```

```
    const { userId, email } = event;

    // Step 1: Create user account
    const user = await context.step("create-account", async () => {
      return await userService.createAccount(userId, email);
    });

    // Step 2: Send verification email
    await context.step("send-verification", async () => {
      return await emailService.sendVerification(email);
    });

    // Step 3: Wait for email verification (up to 48 hours)
    const verified = await context.waitForCallback(
      "email-verification",
      async (callbackId) => {
        await notificationService.sendVerificationLink({
          email,
          callbackId,
          expiresIn: 172800
        });
      },
      {
        timeout: { seconds: 172800 }
      }
    );

    if (!verified) {
      await context.step("send-reminder", async () => {
        await emailService.sendReminder(email);
      });

      return {
        status: "verification_timeout",
        userId,
        message: "Email verification not completed within 48 hours"
      };
    }

    // Step 4: Initialize user profile in parallel
    const setupResults = await context.parallel("profile-setup", [
      async (ctx: DurableContext) => {
        return await ctx.step("create-preferences", async () => {
          return await preferencesService.createDefaults(userId);
```

```
        });
      },

      async (ctx: DurableContext) => {
        return await ctx.step("setup-notifications", async () => {
          return await notificationService.setupDefaults(userId);
        });
      },

      async (ctx: DurableContext) => {
        return await ctx.step("create-welcome-content", async () => {
          return await contentService.createWelcome(userId);
        });
      }
    ]);

    // Step 5: Send welcome email
    await context.step("send-welcome", async () => {
      const [preferences, notifications, content] = setupResults.getResults();
      return await emailService.sendWelcome({
        email,
        preferences,
        notifications,
        content
      });
    });

    return {
      status: "onboarding_complete",
      userId,
      completedAt: new Date().toISOString()
    };
  }
);
```

Python

```
from aws_durable_execution_sdk import DurableContext, durable_execution, WaitConfig
from datetime import datetime

@durable_execution
```

```
def lambda_handler(event, context: DurableContext):
    user_id = event['userId']
    email = event['email']

    # Step 1: Create user account
    user = context.step(
        lambda _: user_service.create_account(user_id, email),
        name='create-account'
    )

    # Step 2: Send verification email
    context.step(
        lambda _: email_service.send_verification(email),
        name='send-verification'
    )

    # Step 3: Wait for email verification (up to 48 hours)
    def send_verification_link(callback_id):
        notification_service.send_verification_link({
            'email': email,
            'callbackId': callback_id,
            'expiresIn': 172800
        })

    verified = context.wait_for_callback(
        send_verification_link,
        name='email-verification',
        config=WaitConfig(timeout=172800)
    )

    if not verified:
        context.step(
            lambda _: email_service.send_reminder(email),
            name='send-reminder'
        )

        return {
            'status': 'verification_timeout',
            'userId': user_id,
            'message': 'Email verification not completed within 48 hours'
        }

    # Step 4: Initialize user profile in parallel
    def create_preferences(ctx: DurableContext):
```

```
        return ctx.step(
            lambda _: preferences_service.create_defaults(user_id),
            name='create-preferences'
        )

    def setup_notifications(ctx: DurableContext):
        return ctx.step(
            lambda _: notification_service.setup_defaults(user_id),
            name='setup-notifications'
        )

    def create_welcome_content(ctx: DurableContext):
        return ctx.step(
            lambda _: content_service.create_welcome(user_id),
            name='create-welcome-content'
        )

    setup_results = context.parallel(
        [create_preferences, setup_notifications, create_welcome_content],
        name='profile-setup'
    )

    # Step 5: Send welcome email
    def send_welcome(_):
        results = setup_results.get_results()
        preferences, notifications, content = results[0], results[1], results[2]
        return email_service.send_welcome({
            'email': email,
            'preferences': preferences,
            'notifications': notifications,
            'content': content
        })

    context.step(send_welcome, name='send-welcome')

    return {
        'status': 'onboarding_complete',
        'userId': user_id,
        'completedAt': datetime.now().isoformat()
    }
```

Le processus combine des étapes séquentielles avec des points de contrôle pour la création de comptes et l'envoi d'e-mails, puis s'interrompt pendant 48 heures en attendant la vérification des e-mails sans consommer de ressources. La logique conditionnelle gère différents chemins selon que la vérification est terminée ou expirée. Les tâches de configuration du profil s'exécutent simultanément à l'aide d'opérations parallèles afin de réduire le temps d'exécution total, et chaque étape est réessayée automatiquement en cas d'échec transitoire afin de garantir la fiabilité de l'intégration.

Traitement par lots avec points de contrôle

Traitez des millions d'enregistrements grâce à la restauration automatique depuis le dernier point de contrôle réussi après un échec. Cet exemple montre comment les fonctions durables combinent les `map()` opérations avec le découpage et la limitation de débit pour gérer le traitement de données à grande échelle.

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";

interface Batch {
  batchIndex: number;
  recordIds: string[];
}

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { datasetId, batchSize = 1000 } = event;

    // Step 1: Get all record IDs to process
    const recordIds = await context.step("fetch-record-ids", async () => {
      return await dataService.getRecordIds(datasetId);
    });

    // Step 2: Split into batches
    const batches: Batch[] = [];
    for (let i = 0; i < recordIds.length; i += batchSize) {
      batches.push({
        batchIndex: Math.floor(i / batchSize),
        recordIds: recordIds.slice(i, i + batchSize)
      });
    }
```

```
    // Step 3: Process batches with controlled concurrency
    const batchResults = await context.map(
      "process-batches",
      batches,
      async (ctx: DurableContext, batch: Batch, index: number) => {
        const processed = await ctx.step(`batch-${batch.batchIndex}`, async () => {
          const results = [];
          for (const recordId of batch.recordIds) {
            const result = await recordService.process(recordId);
            results.push(result);
          }
          return results;
        });

        const validated = await ctx.step(`validate-${batch.batchIndex}`, async () =>
{
          return await validationService.validateBatch(processed);
        });

        return {
          batchIndex: batch.batchIndex,
          recordCount: batch.recordIds.length,
          successCount: validated.successCount,
          failureCount: validated.failureCount
        };
      },
      {
        maxConcurrency: 5
      }
    );

    // Step 4: Aggregate results
    const summary = await context.step("aggregate-results", async () => {
      const results = batchResults.getResults();
      const totalSuccess = results.reduce((sum, r) => sum + r.successCount, 0);
      const totalFailure = results.reduce((sum, r) => sum + r.failureCount, 0);

      return {
        datasetId,
        totalRecords: recordIds.length,
        batchesProcessed: batches.length,
        successCount: totalSuccess,
        failureCount: totalFailure,
```

```
        completedAt: new Date().toISOString()
      };
    });

    return summary;
  }
);
```

Python

```
from aws_durable_execution_sdk import DurableContext, durable_execution, MapConfig
from datetime import datetime
from typing import List, Dict

@durable_execution
def lambda_handler(event, context: DurableContext):
    dataset_id = event['datasetId']
    batch_size = event.get('batchSize', 1000)

    # Step 1: Get all record IDs to process
    record_ids = context.step(
        lambda _: data_service.get_record_ids(dataset_id),
        name='fetch-record-ids'
    )

    # Step 2: Split into batches
    batches = []
    for i in range(0, len(record_ids), batch_size):
        batches.append({
            'batchIndex': i // batch_size,
            'recordIds': record_ids[i:i + batch_size]
        })

    # Step 3: Process batches with controlled concurrency
    def process_batch(ctx: DurableContext, batch: Dict, index: int):
        batch_index = batch['batchIndex']

        def process_records(_):
            results = []
            for record_id in batch['recordIds']:
                result = record_service.process(record_id)
```

```
                results.append(result)
            return results

        processed = ctx.step(process_records, name=f'batch-{batch_index}')

        validated = ctx.step(
            lambda _: validation_service.validate_batch(processed),
            name=f'validate-{batch_index}'
        )

        return {
            'batchIndex': batch_index,
            'recordCount': len(batch['recordIds']),
            'successCount': validated['successCount'],
            'failureCount': validated['failureCount']
        }

    batch_results = context.map(
        process_batch,
        batches,
        name='process-batches',
        config=MapConfig(max_concurrency=5)
    )

    # Step 4: Aggregate results
    def aggregate_results(_):
        results = batch_results.get_results()
        total_success = sum(r['successCount'] for r in results)
        total_failure = sum(r['failureCount'] for r in results)

        return {
            'datasetId': dataset_id,
            'totalRecords': len(record_ids),
            'batchesProcessed': len(batches),
            'successCount': total_success,
            'failureCount': total_failure,
            'completedAt': datetime.now().isoformat()
        }

    summary = context.step(aggregate_results, name='aggregate-results')

    return summary
```

Les enregistrements sont divisés en lots gérables pour éviter de surcharger la mémoire ou les services en aval, puis plusieurs lots sont traités simultanément avec `maxConcurrency` le contrôle du parallélisme. Chaque lot possède son propre point de contrôle, de sorte que les échecs ne font que réessayer le lot défaillant au lieu de retraiter tous les enregistrements. Ce modèle est idéal pour les tâches ETL, les migrations de données ou les opérations en masse dont le traitement peut prendre des heures.

Étapes suivantes

- Explorez les concepts de base pour comprendre DurableContext, les étapes et les attentes
- Passez en revue les meilleures pratiques pour écrire du code déterministe et optimiser les performances
- Découvrez comment tester des fonctions durables en local et dans le cloud

Sécurité et autorisations pour les fonctions durables de Lambda

Les fonctions durables de Lambda nécessitent des autorisations IAM spécifiques pour gérer les opérations des points de contrôle. Respectez le principe du moindre privilège en n'accordant que les autorisations dont votre fonction a besoin.

Autorisations du rôle d'exécution

Le rôle d'exécution de votre fonction durable nécessite des autorisations pour créer des points de contrôle et récupérer l'état d'exécution. La politique suivante indique les autorisations minimales requises :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:CheckpointDurableExecution",
                "lambda:GetDurableExecutionState"
            ],
            "Resource": "arn:aws:lambda:region:account-id:function:function-name:*"
        }
    ]
```

```
}
```

Lorsque vous créez une fonction durable à l'aide de la console, Lambda ajoute automatiquement ces autorisations au rôle d'exécution. Si vous créez la fonction à l'aide du AWS CLI ou AWS CloudFormation, ajoutez ces autorisations à votre rôle d'exécution.

Principe du moindre privilège

Délimitez l'`Resource`élément à une fonction spécifique ARNs au lieu d'utiliser des caractères génériques. Cela limite le rôle d'exécution aux opérations de point de contrôle uniquement pour les fonctions qui en ont besoin.

Exemple : autorisations étendues pour plusieurs fonctions

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:CheckpointDurableExecution",
                "lambda:GetDurableExecutionState"
            ],
            "Resource": [
                "arn:aws:lambda:us-east-1:123456789012:function:orderProcessor:*",
                "arn:aws:lambda:us-east-1:123456789012:function:paymentHandler:*"
            ]
        }
    ]
}
```

Vous pouvez également utiliser la politique AWS gérée `AWSLambdaBasicDurableExecutionRolePolicy` qui inclut les autorisations d'exécution durables requises ainsi que les autorisations d'exécution Lambda de base pour Amazon CloudWatch Logs.

Chiffrement d'état

Les fonctions Lambda Durable activent automatiquement le chiffrement au repos à l'aide de clés AWS détenues gratuitement. Chaque exécution de fonction conserve un état isolé auquel les autres

exécutions ne peuvent pas accéder. Les clés gérées par le client (CMK) ne sont pas prises en charge.

Les données des points de contrôle incluent :

- Résultats des étapes et valeurs renvoyées
- Progrès et calendrier de l'exécution
- Informations sur l'état d'attente

Toutes les données sont cryptées en transit à l'aide du protocole TLS lorsque Lambda lit ou écrit des données de point de contrôle.

Chiffrement personnalisé avec sérialiseurs et désérialiseurs personnalisés

Pour les exigences de sécurité critiques, vous pouvez implémenter votre propre mécanisme de chiffrement et de déchiffrement à l'aide de sérialiseurs et de désérialiseurs personnalisés () à l'aide d'un SDK durable. SerDer Cette approche vous donne un contrôle total sur les clés de chiffrement et les algorithmes utilisés pour protéger les données des points de contrôle.

> **Important**
>
> Lorsque vous utilisez un chiffrement personnalisé, vous perdez la visibilité des résultats des opérations dans la console Lambda et les réponses de l'API. Les données des points de contrôle apparaissent cryptées dans l'historique des exécutions et ne peuvent pas être inspectées sans déchiffrement.

Le rôle d'exécution de votre fonction `kms:Decrypt` nécessite `kms:Encrypt` et autorise la AWS KMS clé utilisée dans l' SerDer implémentation personnalisée.

CloudTrail journalisation

Lambda enregistre les opérations des points de contrôle sous forme d'événements de données dans. AWS CloudTrail Vous pouvez l'utiliser CloudTrail pour auditer le moment où les points de contrôle sont créés, suivre les changements d'état d'exécution et surveiller l'accès aux données d'exécution durables.

Les opérations de point de contrôle apparaissent dans CloudTrail les journaux sous les noms d'événements suivants :

- `CheckpointDurableExecution`- Enregistré lorsqu'une étape se termine et crée un point de contrôle
- `GetDurableExecutionState`- Enregistré lorsque Lambda récupère l'état d'exécution pendant la rediffusion

Pour activer la journalisation des événements de données pour des fonctions durables, configurez un journal CloudTrail pour enregistrer les événements de données Lambda. Pour plus d'informations, consultez la section Journalisation des événements liés aux données dans le Guide de CloudTrail l'utilisateur.

Exemple : entrée de CloudTrail journal pour le fonctionnement du point de contrôle

```
{
    "eventVersion": "1.08",
    "eventTime": "2024-11-16T10:30:45Z",
    "eventName": "CheckpointDurableExecution",
    "eventSource": "lambda.amazonaws.com",
    "requestParameters": {
        "functionName": "myDurableFunction",
        "executionId": "exec-abc123",
        "stepId": "step-1"
    },
    "responseElements": null,
    "eventType": "AwsApiCall"
}
```

Considérations relatives à l’accès intercompte

Si vous invoquez des fonctions durables sur plusieurs AWS comptes, le compte appelant a besoin d'une `lambda:InvokeFunction` autorisation, mais les opérations de point de contrôle utilisent toujours le rôle d'exécution dans le compte de la fonction. Le compte appelant ne peut pas accéder directement aux données du point de contrôle ou à l'état d'exécution.

Cette isolation garantit que les données des points de contrôle restent sécurisées dans le compte de la fonction, même lorsqu'elles sont invoquées depuis des comptes externes.

Fonctionnalités de sécurité Lambda héritées

Les fonctions durables héritent de toutes les fonctionnalités de sécurité, de gouvernance et de conformité de Lambda, notamment la connectivité VPC, le chiffrement des variables

d'environnement, les files d'attente lettre morte, la simultanéité réservée, les URLs fonctions, la signature de code et les certifications de conformité (SOC, PCI DSS, HIPAA, etc.).

Pour des informations détaillées sur les fonctionnalités de sécurité Lambda, consultez la section Sécurité AWS Lambda dans le guide du développeur Lambda. Les seules considérations de sécurité supplémentaires pour les fonctions durables sont les autorisations de point de contrôle décrites dans ce guide.

SDK d'exécution durable

Le SDK d'exécution durable est la base de la création de fonctions durables. Il fournit les primitives dont vous avez besoin pour contrôler la progression, gérer les nouvelles tentatives et gérer le flux d'exécution. Le SDK élimine la complexité de la gestion et de la rediffusion des points de contrôle, en vous permettant d'écrire du code séquentiel qui devient automatiquement tolérant aux pannes.

Le SDK est disponible pour JavaScript, TypeScript, et Python. Pour obtenir une documentation complète sur l'API et des exemples, consultez le TypeScript SDKJavaScript/et le SDK Python sur. GitHub

DurableContext

Le SDK fournit à votre fonction un `DurableContext` objet qui expose toutes les opérations durables. Ce contexte remplace le contexte Lambda standard et fournit des méthodes pour créer des points de contrôle, gérer le flux d'exécution et coordonner avec des systèmes externes.

Pour utiliser le SDK, enveloppez votre gestionnaire Lambda avec le wrapper d'exécution durable :

TypeScript

```
import { withDurableExecution, DurableContext } from '@aws/durable-execution-sdk-
js';

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    // Your function receives DurableContext instead of Lambda context
    // Use context.step(), context.wait(), etc.
    return result;
  }
);
```

Python

```
from aws_durable_execution_sdk_python import durable_execution, DurableContext

@durable_execution
def handler(event: dict, context: DurableContext):
    # Your function receives DurableContext
    # Use context.step(), context.wait(), etc.
    return result
```

Le wrapper intercepte l'invocation de votre fonction, charge tout journal de points de contrôle existant et fournit le journal `DurableContext` qui gère les rediffusions et les points de contrôle.

Ce que fait le SDK

Le SDK assume trois responsabilités essentielles qui permettent une exécution durable :

Gestion des points de contrôle : le SDK crée automatiquement des points de contrôle lorsque votre fonction exécute des opérations durables. Chaque point de contrôle enregistre le type d'opération, les entrées et les résultats. Lorsque votre fonction termine une étape, le SDK maintient le point de contrôle avant de continuer. Cela garantit que votre fonction peut reprendre après toute opération terminée en cas d'interruption.

Coordination des rediffusions : lorsque votre fonction reprend après une pause ou une interruption, le SDK effectue une rediffusion. Il exécute votre code depuis le début mais ignore les opérations terminées, en utilisant les résultats des points de contrôle stockés au lieu de les réexécuter. Le SDK garantit que la rediffusion est déterministe : avec les mêmes entrées et le même journal de points de contrôle, votre fonction produit les mêmes résultats.

Isolation des états : le SDK conserve l'état d'exécution séparément de votre logique métier. Chaque exécution durable possède son propre journal de points de contrôle auquel les autres exécutions ne peuvent pas accéder. Le SDK chiffre les données des points de contrôle au repos et garantit la cohérence de l'état entre les rediffusions.

Comment fonctionne le point de contrôle

Lorsque vous appelez une opération durable, le SDK suit cette séquence :

1. Vérifier l'existence d'un point de contrôle : le SDK vérifie si cette opération a déjà été effectuée lors d'un appel précédent. S'il existe un point de contrôle, le SDK renvoie le résultat enregistré sans réexécuter l'opération.
2. Exécuter l'opération : s'il n'existe aucun point de contrôle, le SDK exécute votre code d'opération. Pour les étapes, cela signifie appeler votre fonction. Pour les temps d'attente, cela signifie planifier la reprise.
3. Créer un point de contrôle : une fois l'opération terminée, le SDK sérialise le résultat et crée un point de contrôle. Le point de contrôle inclut le type d'opération, le nom, les entrées, le résultat et l'horodatage.
4. Point de contrôle persistant : le SDK appelle l'API du point de contrôle Lambda pour conserver le point de contrôle. Cela garantit la durabilité du point de contrôle avant de poursuivre l'exécution.
5. Renvoyer le résultat : le SDK renvoie le résultat de l'opération à votre code, qui passe à l'opération suivante.

Cette séquence garantit qu'une fois l'opération terminée, son résultat est stocké en toute sécurité. Si votre fonction est interrompue à tout moment, le SDK peut être rejoué jusqu'au dernier point de contrôle terminé.

Comportement de rediffusion

Lorsque votre fonction reprend après une pause ou une interruption, le SDK effectue une rediffusion :

1. Charger le journal des points de contrôle : le SDK extrait le journal des points de contrôle pour cette exécution à partir de Lambda.
2. Exécuter depuis le début : le SDK appelle votre fonction de gestionnaire dès le début, et non depuis l'endroit où elle s'est interrompue.
3. Ignorer les opérations durables terminées : lorsque votre code appelle des opérations durables, le SDK compare chacune d'entre elles au journal des points de contrôle. Pour les opérations durables terminées, le SDK renvoie le résultat stocké sans exécuter le code d'opération.

> **Note**
>
> Si le résultat d'un contexte enfant est supérieur à la taille maximale du point de contrôle (256 Ko), le code du contexte est à nouveau exécuté pendant la rediffusion. Cela vous permet de générer des résultats volumineux à partir des opérations durables exécutées dans le contexte, qui seront consultés dans le journal des points de contrôle. Il est donc

impératif de n'exécuter le code déterministe que dans le contexte lui-même. Lorsque vous utilisez des contextes enfants avec des résultats importants, il est recommandé d'effectuer un travail de longue durée ou non déterministe à l'intérieur des étapes et de n'effectuer que des tâches de courte durée qui combinent les résultats dans le contexte lui-même.

4. Reprise au point d'interruption : lorsque le SDK atteint une opération sans point de contrôle, il s'exécute normalement et crée de nouveaux points de contrôle au fur et à mesure que les opérations durables sont terminées.

Ce mécanisme de rediffusion nécessite que votre code soit déterministe. Avec les mêmes entrées et le même journal de points de contrôle, votre fonction doit effectuer la même séquence d'appels d'opérations durables. Le SDK applique cela en validant que les noms et types d'opérations correspondent au journal des points de contrôle lors de la rediffusion.

Opérations durables disponibles

`DurableContext`Il fournit des opérations pour différents modèles de coordination. Chaque opération durable crée automatiquement des points de contrôle, garantissant ainsi que votre fonction peut reprendre à tout moment.

Étapes

Exécute la logique métier avec pointage automatique des points de contrôle et réessais. Utilisez des étapes pour les opérations qui appellent des services externes, effectuent des calculs ou exécutent toute logique qui doit être vérifiée. Le SDK crée un point de contrôle avant et après l'étape, stockant le résultat pour le rejouer.

TypeScript

```
const result = await context.step('process-payment', async () => {
  return await paymentService.charge(amount);
});
```

Python

```
result = context.step(
```

```
    lambda _: payment_service.charge(amount),
    name='process-payment'
)
```

Les étapes prennent en charge les stratégies de nouvelle tentative configurables, la sémantique d'exécution (at-most-once ou at-least-once) et la sérialisation personnalisée.

Éléments d’attente

Suspend l'exécution pendant une durée spécifiée sans consommer de ressources informatiques. Le SDK crée un point de contrôle, met fin à l'invocation de la fonction et planifie la reprise. Lorsque l'attente est terminée, Lambda invoque à nouveau votre fonction et le SDK rejoue jusqu'au point d'attente avant de continuer.

TypeScript

```
// Wait 1 hour without charges
await context.wait({ seconds: 3600 });
```

Python

```
# Wait 1 hour without charges
context.wait(3600)
```

Rappels

Les rappels permettent à votre fonction de faire une pause et d'attendre que des systèmes externes fournissent une entrée. Lorsque vous créez un rappel, le SDK génère un identifiant de rappel unique et crée un point de contrôle. Votre fonction est ensuite suspendue (met fin à l'invocation) sans frais de calcul. Les systèmes externes soumettent les résultats de rappel à l'aide du `SendDurableExecutionCallbackSuccess` ou `SendDurableExecutionCallbackFailure` Lambda APIs. Lorsqu'un rappel est soumis, Lambda invoque à nouveau votre fonction, le SDK rejoue jusqu'au point de rappel et votre fonction continue avec le résultat du rappel.

Le SDK propose deux méthodes pour utiliser les rappels :

CreateCallback : crée un rappel et renvoie à la fois une promesse et un identifiant de rappel. Vous envoyez l'ID de rappel à un système externe, qui soumet le résultat à l'aide de l'API Lambda.

TypeScript

```
const [promise, callbackId] = await context.createCallback('approval', {
  timeout: { hours: 24 }
});

await sendApprovalRequest(callbackId, requestData);
const approval = await promise;
```

Python

```
callback = context.create_callback(
    name='approval',
    config=CallbackConfig(timeout_seconds=86400)
)

context.step(
    lambda _: send_approval_request(callback.callback_id),
    name='send_request'
)

approval = callback.result()
```

waitForCallback: simplifie la gestion des rappels en combinant la création et la soumission des rappels en une seule opération. Le SDK crée le rappel, exécute votre fonction de soumission avec l'ID de rappel et attend le résultat.

TypeScript

```
const result = await context.waitForCallback(
  'external-api',
```

```
  async (callbackId, ctx) => {
    await submitToExternalAPI(callbackId, requestData);
  },
  { timeout: { minutes: 30 } }
);
```

Python

```
result = context.wait_for_callback(
    lambda callback_id: submit_to_external_api(callback_id, request_data),
    name='external-api',
    config=WaitForCallbackConfig(timeout_seconds=1800)
)
```

Configurez les délais d'attente pour empêcher les fonctions d'attendre indéfiniment. Si un rappel expire, le SDK renvoie un `CallbackError` et votre fonction peut gérer le cas de délai d'expiration. Utilisez les délais d'expiration du rythme cardiaque pour les rappels de longue durée afin de détecter les cas où les systèmes externes cessent de répondre.

Utilisez des rappels pour les human-in-the-loop flux de travail, l'intégration de systèmes externes, les réponses Webhook ou tout autre scénario dans lequel l'exécution doit être interrompue pour une entrée externe.

Exécution en parallèle

Exécute plusieurs opérations simultanément avec un contrôle de simultanéité optionnel. Le SDK gère l'exécution parallèle, crée des points de contrôle pour chaque opération et gère les défaillances conformément à votre politique d'achèvement.

TypeScript

```
const results = await context.parallel([
  async (ctx) => ctx.step('task1', async () => processTask1()),
  async (ctx) => ctx.step('task2', async () => processTask2()),
  async (ctx) => ctx.step('task3', async () => processTask3())
]);
```

Python

```
results = context.parallel(
    lambda ctx: ctx.step(lambda _: process_task1(), name='task1'),
    lambda ctx: ctx.step(lambda _: process_task2(), name='task2'),
    lambda ctx: ctx.step(lambda _: process_task3(), name='task3')
)
```

`parallel`À utiliser pour exécuter simultanément des opérations indépendantes.

Map

Exécutez simultanément une opération sur chaque élément d'un tableau avec un contrôle de simultanéité optionnel. Le SDK gère les exécutions simultanées, crée des points de contrôle pour chaque opération et gère les défaillances conformément à votre politique d'achèvement.

TypeScript

```
const results = await context.map(itemArray, async (ctx, item, index) =>
  ctx.step('task', async () => processItem(item, index))
);
```

Python

```
results = context.map(
    item_array,
    lambda ctx, item, index: ctx.step(
        lambda _: process_item(item, index),
        name='task'
    )
)
```

`map`À utiliser pour traiter des tableaux avec contrôle de simultanéité.

Contextes relatifs aux enfants

Crée un contexte d'exécution isolé pour les opérations de regroupement. Les contextes enfants possèdent leur propre journal des points de contrôle et peuvent contenir plusieurs étapes, temps d'attente et autres opérations. Le SDK traite l'ensemble du contexte de l'enfant comme une unité unique pour les nouvelles tentatives et la restauration.

Utilisez des contextes enfants pour organiser des flux de travail complexes, implémenter des sous-flux de travail ou isoler les opérations qui doivent être réessayées ensemble.

TypeScript

```
const result = await context.runInChildContext(
  'batch-processing',
  async (childCtx) => {
    return await processBatch(childCtx, items);
  }
);
```

Python

```
result = context.run_in_child_context(
    lambda child_ctx: process_batch(child_ctx, items),
    name='batch-processing'
)
```

Le mécanisme de rediffusion exige que les opérations durables soient effectuées dans un ordre déterministe. En utilisant plusieurs contextes enfants, plusieurs flux de travail peuvent être exécutés simultanément, et le déterminisme s'applique séparément dans chaque contexte. Cela vous permet de créer des fonctions hautes performances qui utilisent efficacement plusieurs cœurs de processeur.

Par exemple, imaginons que nous démarrions deux contextes enfants, A et B. Lors de l'invocation initiale, les étapes des contextes étaient exécutées dans cet ordre, les étapes « A » s'exécutant simultanément avec les étapes « B » : A1, B1, B2, A2, A3. Lors de la rediffusion, le chronométrage

est beaucoup plus rapide car les résultats sont extraits du journal des points de contrôle et les étapes sont parcourues dans un ordre différent : B1, A1, A2, B2, A3. Comme les étapes « A » ont été rencontrées dans le bon ordre (A1, A2, A3) et que les étapes « B » ont été rencontrées dans le bon ordre (B1, B2), le besoin de déterminisme a été correctement satisfait.

Attentes conditionnelles

Recherche une condition avec pointage automatique entre les tentatives. Le SDK exécute votre fonction de vérification, crée un point de contrôle avec le résultat, attend selon votre stratégie et répète jusqu'à ce que la condition soit remplie.

TypeScript

```
const result = await context.waitForCondition(
  async (state, ctx) => {
    const status = await checkJobStatus(state.jobId);
    return { ...state, status };
  },
  {
    initialState: { jobId: 'job-123', status: 'pending' },
    waitStrategy: (state) =>
      state.status === 'completed'
        ? { shouldContinue: false }
        : { shouldContinue: true, delay: { seconds: 30 } }
  }
);
```

Python

```
result = context.wait_for_condition(
    lambda state, ctx: check_job_status(state['jobId']),
    config=WaitForConditionConfig(
        initial_state={'jobId': 'job-123', 'status': 'pending'},
        wait_strategy=lambda state, attempt:
            {'should_continue': False} if state['status'] == 'completed'
            else {'should_continue': True, 'delay': 30}
    )
)
```

`waitForCondition`À utiliser pour interroger des systèmes externes, attendre que les ressources soient prêtes ou implémenter une nouvelle tentative avec interruption.

Invocation de fonctions

Invoque une autre fonction Lambda et attend son résultat. Le SDK crée un point de contrôle, invoque la fonction cible et reprend votre fonction une fois l'invocation terminée. Cela permet la composition des fonctions et la décomposition du flux de travail.

TypeScript

```
const result = await context.invoke(
  'invoke-processor',
  'arn:aws:lambda:us-east-1:123456789012:function:processor',
  { data: inputData }
);
```

Python

```
result = context.invoke(
    'arn:aws:lambda:us-east-1:123456789012:function:processor',
    {'data': input_data},
    name='invoke-processor'
)
```

Comment mesure-t-on la durabilité des opérations

Chaque opération durable que vous effectuez `DurableContext` crée des points de contrôle pour suivre la progression de l'exécution et stocker les données d'état. Ces opérations sont facturées en fonction de leur utilisation, et les points de contrôle peuvent contenir des données qui contribuent à vos coûts d'écriture et de conservation des données. Les données stockées incluent les données relatives aux événements d'invocation, les charges utiles renvoyées par les étapes et les données transmises lors de l'exécution des rappels. Comprendre comment la durabilité des opérations est mesurée vous permet d'estimer les coûts d'exécution et d'optimiser vos flux de travail. Pour plus de détails sur la tarification, consultez la page de tarification de Lambda.

La taille de la charge utile fait référence à la taille des données sérialisées qu'une opération durable persiste. Les données sont mesurées en octets et leur taille peut varier en fonction du sérialiseur utilisé par l'opération. La charge utile d'une opération peut être le résultat lui-même en cas d'achèvement réussi, ou l'objet d'erreur sérialisé en cas d'échec de l'opération.

Opérations de base

Les opérations de base sont les éléments fondamentaux des fonctions durables :

Opération	Chronométrage des points de contrôle	Nombre d'opérations	Les données ont persisté
Exécution	Démarré(e)	1	Taille de la charge utile d'entrée
Exécution	Terminé (Succeeded/Failed/Stopped)	0	Taille de la charge utile de sortie
Step (Étape)	Retry/Succeeded/Failed	10 + Aucune nouvelle tentative	Taille de la charge utile renvoyée à chaque tentative
Attente	Démarré(e)	1	N/A
WaitForCondition	Chaque tentative de sondage	+ 1 x N sondages	Taille de la charge utile renvoyée à chaque tentative de sondage
Nouvelle tentative au niveau de l'invocation	Démarré(e)	1	Charge utile pour l'objet d'erreur

Opérations de rappel

Les opérations de rappel permettent à votre fonction de faire une pause et d'attendre que des systèmes externes fournissent une entrée. Ces opérations créent des points de contrôle lorsque le rappel est créé et lorsqu'il est terminé :

Opération	Chronométrage des points de contrôle	Nombre d'opérations	Les données ont persisté
CreateCallback	Démarré(e)	1	N/A
Exécution du rappel via un appel d'API	Terminé	0	Charge utile de rappel
WaitForCallback	Démarré(e)	3 + N tentatives (contexte + rappel + étape)	Charges utiles renvoyées par les tentatives d'étape de l'expéditeur, plus deux copies de la charge utile de rappel

Opérations composées

Les opérations composées combinent plusieurs opérations durables pour gérer des modèles de coordination complexes tels que l'exécution parallèle, le traitement de tableaux et les contextes imbriqués :

Opération	Chronométrage des points de contrôle	Nombre d'opérations	Les données ont persisté
Parallèle	Démarré(e)	1 + N branches (1 contexte parent + N contextes enfants)	Jusqu'à deux copies de la taille de charge utile renvoyée par chaque branche, plus les statuts de chaque branche
Map	Démarré(e)	1 + N branches (1	Jusqu'à deux copies de la taille de charge utile renvoyée par itération, plus les statuts de chaque itération

Opération	Chronométrage des points de contrôle	Nombre d'opérations	Les données ont persisté
		contexte parent + N contextes enfants)	
Des aides à la promesse	Terminé	1	Taille de la charge utile renvoyée par la promesse
RunInChildContext	Succès/Echec	1	Taille de la charge utile renvoyée par le contexte enfant

Pour les contextes, tels que ceux issus `runInChildContext` ou utilisés en interne par des opérations composées, les résultats inférieurs à 256 Ko sont directement contrôlés. Les résultats les plus importants ne sont pas stockés ; ils sont reconstruits lors de la rediffusion en retraitant les opérations du contexte.

Runtimes pris en charge pour des fonctions durables

Des fonctions durables sont disponibles pour les environnements d'exécution Node.js et Python. Vous pouvez créer des fonctions durables à l'aide d'environnements d'exécution gérés dans la console Lambda ou les déployer à l'aide d'images de conteneur pour une flexibilité accrue des versions d'exécution.

Runtimes gérés par Lambda

Les environnements d'exécution gérés suivants prennent en charge les fonctions durables lorsque vous créez des fonctions dans la console Lambda ou que vous utilisez AWS CLI le paramètre with `--durable-config '{"ExecutionTimeout": 10, "RetentionPeriodInDays":1}'` the. Pour des informations complètes sur les environnements d'exécution Lambda, consultez la section Runtimes Lambda.

Language	Environnement d’exécution
Node.js	nodejs22.x
Node.js	nodejs24.x
Python	python3.13
Python	python3,14

> **Note**
>
> Les environnements d'exécution Lambda incluent le SDK d'exécution durable pour les tests et le développement. Toutefois, nous vous recommandons d'inclure le SDK dans votre package de déploiement pour la production. Cela garantit la cohérence des versions et évite les mises à jour d'exécution potentielles susceptibles d'affecter le comportement de votre fonction.

Node.js

Installez le SDK dans votre projet Node.js :

```
npm install @aws/durable-execution-sdk-js
```

Le SDK prend en charge JavaScript et. TypeScript Pour les TypeScript projets, le SDK inclut des définitions de type.

Python

Installez le SDK dans votre projet Python :

```
pip install aws-durable-execution-sdk-python
```

Le SDK Python utilise des méthodes synchrones et n'en nécessite pas. `async/await`

Images de conteneurs

Vous pouvez utiliser des fonctions durables avec des images de conteneur pour prendre en charge des versions d'exécution supplémentaires ou des configurations d'exécution personnalisées. Les images de conteneur vous permettent d'utiliser des versions d'exécution non disponibles en tant que moteurs d'exécution gérés ou de personnaliser votre environnement d'exécution.

Pour créer une fonction durable à l'aide d'une image de conteneur :

1. Création d'un Dockerfile basé sur une image de base Lambda
2. Installez le SDK d'exécution durable dans votre conteneur
3. Créez et transférez l'image du conteneur vers Amazon Elastic Container Registry
4. Créez la fonction Lambda à partir de l'image du conteneur avec l'exécution durable activée

Exemple de conteneur Python

Créez un Dockerfile pour Python 3.11 :

```
FROM public.ecr.aws/lambda/python:3.11

# Copy requirements file
COPY requirements.txt ${LAMBDA_TASK_ROOT}/

# Install dependencies including durable SDK
RUN pip install -r requirements.txt

# Copy function code
COPY lambda_function.py ${LAMBDA_TASK_ROOT}/

# Set the handler
CMD [ "lambda_function.handler" ]
```

Créez un `requirements.txt` fichier :

```
aws-durable-execution-sdk-python
```

Créez et publiez l'image :

```
# Build the image
docker build -t my-durable-function .

# Tag for ECR
docker tag my-durable-function:latest 123456789012.dkr.ecr.us-east-1.amazonaws.com/my-
durable-function:latest

# Push to ECR
docker push 123456789012.dkr.ecr.us-east-1.amazonaws.com/my-durable-function:latest
```

Créez la fonction avec l'exécution durable activée :

```
aws lambda create-function \
  --function-name myDurableFunction \
  --package-type Image \
  --code ImageUri=123456789012.dkr.ecr.us-east-1.amazonaws.com/my-durable-
function:latest \
  --role arn:aws:iam::123456789012:role/lambda-execution-role \
  --durable-config '{"ExecutionTimeout": 10, "RetentionPeriodInDays":1}'
```

Pour plus d'informations sur l'utilisation d'images de conteneur avec Lambda, consultez la section Création d'images de conteneur Lambda dans le guide du développeur Lambda.

Considérations relatives à l’exécution

Gestion des versions du SDK : incluez le SDK d'exécution durable dans votre package de déploiement ou votre image de conteneur. Cela garantit que votre fonction utilise une version spécifique du SDK et n'est pas affectée par les mises à jour d'exécution. Épinglez les versions du SDK dans votre `package.json` ou `requirements.txt` pour contrôler lors de la mise à niveau.

Mises à jour d'exécution : AWS met à jour les environnements d'exécution gérés pour inclure des correctifs de sécurité et des corrections de bogues. Ces mises à jour peuvent inclure de nouvelles versions du SDK. Pour éviter tout comportement inattendu, incluez le SDK dans votre package de déploiement et testez-le de manière approfondie avant de le déployer en production.

Taille de l'image du conteneur : les images du conteneur ont une taille non compressée maximale de 10 Go. Le SDK d'exécution durable ajoute une taille minimale à votre image. Optimisez votre conteneur en utilisant des builds en plusieurs étapes et en supprimant les dépendances inutiles.

Performances de démarrage à froid : les images de conteneur peuvent avoir des temps de démarrage à froid plus longs que les temps d'exécution gérés. Le SDK d'exécution durable a un impact minimal sur les performances de démarrage à froid. Utilisez la simultanéité provisionnée si la latence de démarrage à froid est essentielle pour votre application.

Invoquer des fonctions Lambda durables

Les fonctions Lambda durables prennent en charge les mêmes méthodes d'appel que les fonctions Lambda standard. Vous pouvez appeler des fonctions durables de manière synchrone, asynchrone ou via des mappages de sources d'événements. Le processus d'invocation est identique aux fonctions standard, mais les fonctions durables offrent des fonctionnalités supplémentaires pour les exécutions de longue durée et la gestion automatique des états.

Méthodes d'invocation

Invocation synchrone : invoquez une fonction durable et attendez la réponse. Les appels synchrones sont limités par le Lambda à 15 minutes (ou moins, selon la fonction configurée et le délai d'exécution). Utilisez l'invocation synchrone lorsque vous avez besoin de résultats immédiats ou lorsque vous intégrez APIs des services qui attendent une réponse. Vous pouvez utiliser les opérations d'attente pour un calcul efficace sans perturber l'appelant : l'invocation attend la fin de l'exécution durable complète. Pour les démarrages d'exécution idempotents, utilisez le paramètre de nom d'exécution tel que décrit dans Idempotentie.

```
aws lambda invoke \
  --function-name my-durable-function:1 \
  --cli-binary-format raw-in-base64-out \
  --payload '{"orderId": "12345"}' \
  response.json
```

Invocation asynchrone : mettez en file d'attente un événement pour qu'il soit traité sans attendre de réponse. Lambda place l'événement dans une file d'attente et y retourne immédiatement. Les invocations asynchrones prennent en charge des durées d'exécution allant jusqu'à 1 an. Utilisez

l'invocation asynchrone pour les fire-and-forget scénarios ou lorsque le traitement peut avoir lieu en arrière-plan. Pour les démarrages d'exécution idempotents, utilisez le paramètre de nom d'exécution tel que décrit dans Idempotentie.

```
aws lambda invoke \
  --function-name my-durable-function:1 \
  --invocation-type Event \
  --cli-binary-format raw-in-base64-out \
  --payload '{"orderId": "12345"}' \
  response.json
```

Mappages de sources d'événements : configurez Lambda pour appeler automatiquement votre fonction durable lorsque des enregistrements sont disponibles à partir de services basés sur des flux ou des files d'attente tels qu'Amazon SQS, Kinesis ou DynamoDB. Les mappages de sources d'événements interrogent la source d'événements et invoquent votre fonction avec des lots d'enregistrements. Pour plus de détails sur l'utilisation des mappages de sources d'événements avec des fonctions durables, y compris les limites de durée d'exécution, voir Mappages de sources d'événements avec des fonctions durables.

Pour obtenir des informations complètes sur chaque méthode d'appel, consultez les sections Invocation synchrone et Invocation asynchrone.

> **Note**
>
> Les fonctions durables prennent en charge les files d'attente en lettres mortes (DLQs) pour la gestion des erreurs, mais ne prennent pas en charge les destinations Lambda. Configurez un DLQ pour capturer les enregistrements des appels ayant échoué.

ARNs Exigence qualifiée

Les fonctions durables nécessitent des identifiants qualifiés pour l'invocation. Vous devez invoquer des fonctions durables à l'aide d'un numéro de version, d'un alias ou`$LATEST`. Vous pouvez utiliser un ARN qualifié complet ou un nom de fonction avec version/alias suffixe. Vous ne pouvez pas utiliser un identifiant non qualifié (sans version ni suffixe d'alias).

Invocations valides :

```
# Using full ARN with version number
arn:aws:lambda:us-east-1:123456789012:function:my-durable-function:1

# Using full ARN with alias
arn:aws:lambda:us-east-1:123456789012:function:my-durable-function:prod

# Using full ARN with $LATEST
arn:aws:lambda:us-east-1:123456789012:function:my-durable-function:$LATEST

# Using function name with version number
my-durable-function:1

# Using function name with alias
my-durable-function:prod
```

Invocations non valides :

```
# Unqualified ARN (not allowed)
arn:aws:lambda:us-east-1:123456789012:function:my-durable-function

# Unqualified function name (not allowed)
my-durable-function
```

Cette exigence garantit que les exécutions durables restent cohérentes tout au long de leur cycle de vie. Lorsqu'une exécution durable démarre, elle est associée à la version de fonction spécifique. Si votre fonction s'arrête et reprend des heures ou des jours plus tard, Lambda invoque la même version qui a démarré l'exécution, garantissant ainsi la cohérence du code dans l'ensemble du flux de travail.

Bonne pratique

Utilisez plutôt `$LATEST` des versions numérotées ou des alias pour produire des fonctions durables. Les versions numérotées sont immuables et garantissent une rediffusion déterministe. Les alias fournissent éventuellement une référence stable que vous pouvez mettre à jour pour pointer vers de nouvelles versions sans modifier le code d'appel. Lorsque vous mettez à jour un alias, les nouvelles exécutions utilisent la nouvelle version, tandis que les exécutions en cours reprennent leur version d'origine. Vous pouvez l'utiliser `$LATEST` pour le prototypage ou pour raccourcir les délais

de déploiement pendant le développement, en sachant que les exécutions risquent de ne pas se reproduire correctement (voire d'échouer) si le code sous-jacent change pendant les exécutions.

Comprendre le cycle de vie d'exécution

Lorsque vous appelez une fonction durable, Lambda crée une exécution durable qui peut couvrir plusieurs invocations de fonction :

1. Invocation initiale : votre demande d'invocation crée une nouvelle exécution durable. Lambda attribue un ID d'exécution unique et lance le traitement.
2. Exécution et pointage : lorsque votre fonction exécute des opérations durables, le SDK crée des points de contrôle qui permettent de suivre la progression.
3. Suspension (si nécessaire) : si votre fonction utilise des temps d'attente prolongés, tels que `wait` des tentatives ou des tentatives automatiques`waitForCallback`, Lambda suspend l'exécution et arrête de facturer le temps de calcul.
4. Reprise : au moment de reprendre (y compris après de nouvelles tentatives), Lambda invoque à nouveau votre fonction. Le SDK rejoue le journal des points de contrôle et continue à partir de l'endroit où l'exécution a été interrompue.
5. Achèvement : lorsque votre fonction renvoie un résultat final ou renvoie une erreur non gérée, l'exécution durable est terminée.

Pour les invocations synchrones, l'appelant attend la fin complète de l'exécution durable, y compris les opérations d'attente. Si l'exécution dépasse le délai d'invocation (15 minutes ou moins), l'invocation expire. Pour les invocations asynchrones, Lambda revient immédiatement et l'exécution se poursuit indépendamment. Utilisez l'exécution durable APIs pour suivre l'état d'exécution et obtenir les résultats finaux.

Invocation depuis le code de l'application

Utilisez le AWS SDKs pour invoquer des fonctions durables à partir du code de votre application. Le processus d'invocation est identique aux fonctions standard :

TypeScript

```
import { LambdaClient, InvokeCommand } from '@aws-sdk/client-lambda';

const client = new LambdaClient({});
```

```
// Synchronous invocation
const response = await client.send(new InvokeCommand({
  FunctionName: 'arn:aws:lambda:us-east-1:123456789012:function:my-durable-
function:1',
  Payload: JSON.stringify({ orderId: '12345' })
}));

const result = JSON.parse(Buffer.from(response.Payload!).toString());

// Asynchronous invocation
await client.send(new InvokeCommand({
  FunctionName: 'arn:aws:lambda:us-east-1:123456789012:function:my-durable-
function:1',
  InvocationType: 'Event',
  Payload: JSON.stringify({ orderId: '12345' })
}));
```

Python

```
import boto3
import json

client = boto3.client('lambda')

# Synchronous invocation
response = client.invoke(
    FunctionName='arn:aws:lambda:us-east-1:123456789012:function:my-durable-
function:1',
    Payload=json.dumps({'orderId': '12345'})
)

result = json.loads(response['Payload'].read())

# Asynchronous invocation
client.invoke(
    FunctionName='arn:aws:lambda:us-east-1:123456789012:function:my-durable-
function:1',
    InvocationType='Event',
    Payload=json.dumps({'orderId': '12345'})
)
```

Invocations enchaînées

Les fonctions durables peuvent invoquer d'autres fonctions durables et non durables à l'aide de l'`invoke`opération de`DurableContext`. Cela crée un appel enchaîné dans lequel la fonction appelante attend (suspend) la fin de la fonction invoquée :

TypeScript

```
export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    // Invoke another durable function and wait for result
    const result = await context.invoke(
      'process-order',
      'arn:aws:lambda:us-east-1:123456789012:function:order-processor:1',
      { orderId: event.orderId }
    );

    return { statusCode: 200, body: JSON.stringify(result) };
  }
);
```

Python

```
@durable_execution
def handler(event, context: DurableContext):
    # Invoke another durable function and wait for result
    result = context.invoke(
        'arn:aws:lambda:us-east-1:123456789012:function:order-processor:1',
        {'orderId': event['orderId']},
        name='process-order'
    )

    return {'statusCode': 200, 'body': json.dumps(result)}
```

Les invocations enchaînées créent un point de contrôle dans la fonction d'appel. Si la fonction appelante est interrompue, elle repart du point de contrôle avec le résultat de la fonction invoquée, sans la réinvoquer.

> **Note**
>
> Les invocations chaînées entre comptes ne sont pas prises en charge. La fonction invoquée doit se trouver dans le même AWS compte que la fonction appelante.

Mappages de sources d'événements dotés de fonctions durables

Les fonctions durables fonctionnent avec tous les mappages de sources d'événements Lambda. Configurez les mappages de sources d'événements pour les fonctions durables de la même manière que vous les configurez pour les fonctions standard. Les mappages de sources d'événements interrogent automatiquement des sources d'événements telles qu'Amazon SQS, Kinesis et DynamoDB Streams, et invoquent votre fonction à l'aide de lots d'enregistrements.

Les mappages de sources d'événements sont utiles pour les fonctions durables qui traitent des flux ou des files d'attente avec des flux de travail complexes en plusieurs étapes. Par exemple, vous pouvez créer une fonction durable qui traite les messages Amazon SQS avec de nouvelles tentatives, des appels d'API externes et des approbations humaines.

Comment les mappages de sources d'événements invoquent des fonctions durables

Les mappages de sources d'événements invoquent des fonctions durables de manière synchrone, en attendant la fin de l'exécution durable complète avant de traiter le lot suivant ou de marquer les enregistrements comme traités. Si le temps d'exécution durable total dépasse 15 minutes, l'exécution expire et échoue. Le mappage de la source d'événements reçoit une exception de délai d'attente et la gère en fonction de sa configuration de nouvelle tentative.

Limite d'exécution de 15 minutes

Lorsque des fonctions durables sont invoquées par des mappages de sources d'événements, la durée totale d'exécution durable ne peut pas dépasser 15 minutes. Cette limite s'applique à l'ensemble de l'exécution durable, du début à la fin, et pas uniquement aux invocations de fonctions individuelles.

Cette limite de 15 minutes est distincte du délai d'expiration de la fonction Lambda (également 15 minutes maximum). La fonction timeout contrôle la durée d'exécution de chaque invocation

individuelle, tandis que le délai d'exécution durable contrôle le temps total écoulé entre le début et la fin de l'exécution.

Exemples de scénarios :

- Valable : une fonction durable traite un message Amazon SQS en trois étapes, chacune prenant 2 minutes, puis attend 5 minutes avant de terminer une dernière étape. Durée totale d'exécution : 11 minutes. Cela fonctionne car le total est inférieur à 15 minutes.
- Non valide : une fonction durable traite un message Amazon SQS, effectue le traitement initial en 2 minutes, puis attend un rappel externe pendant 20 minutes avant de terminer. Durée totale d'exécution : 22 minutes. Cela dépasse la limite de 15 minutes et échouera.
- Non valide : une fonction durable traite un enregistrement Kinesis avec plusieurs opérations d'attente d'une durée totale de 30 minutes entre les étapes. Même si chaque invocation individuelle se termine rapidement, le temps d'exécution total dépasse 15 minutes.

> **Important**
>
> Configurez votre délai d'exécution durable à 15 minutes ou moins lorsque vous utilisez des mappages de sources d'événements, sinon la création du mappage des sources d'événements échouera. Si votre flux de travail nécessite des temps d'exécution plus longs, utilisez le modèle de fonction intermédiaire décrit ci-dessous.

Configuration des mappages de sources d'événements

Configurez les mappages de sources d'événements pour des fonctions durables à l'aide de la console Lambda AWS CLI, ou. AWS SDKs Toutes les propriétés standard de mappage des sources d'événements s'appliquent aux fonctions durables :

```
aws lambda create-event-source-mapping \
  --function-name arn:aws:lambda:us-east-1:123456789012:function:my-durable-function:1
 \
  --event-source-arn arn:aws:sqs:us-east-1:123456789012:my-queue \
  --batch-size 10 \
  --maximum-batching-window-in-seconds 5
```

N'oubliez pas d'utiliser un ARN qualifié (avec numéro de version ou alias) lors de la configuration des mappages de sources d'événements pour des fonctions durables.

Gestion des erreurs avec les mappages de sources d'événements

Les mappages de sources d'événements fournissent une gestion des erreurs intégrée qui fonctionne avec des fonctions durables :

- Comportement de nouvelle tentative : si l'appel initial échoue, le mappage de la source d'événements réessaie conformément à sa configuration de nouvelle tentative. Configurez le nombre maximal de tentatives et les intervalles entre les tentatives en fonction de vos besoins.
- Files d'attente de lettres mortes : configurez une file d'attente de lettres mortes pour capturer les enregistrements qui échouent après chaque nouvelle tentative. Cela empêche la perte de messages et permet une inspection manuelle des enregistrements défaillants.
- Défaillances partielles par lots : pour Amazon SQS et Kinesis, utilisez le rapport d'échec partiel des lots pour traiter les enregistrements individuellement et réessayez uniquement les enregistrements ayant échoué.
- Couper en deux en cas d'erreur : pour Kinesis et DynamoDB Streams, activez la division en deux en cas d'erreur pour diviser les lots ayant échoué et isoler les enregistrements problématiques.

> **Note**
>
> Les fonctions durables prennent en charge les files d'attente en lettres mortes (DLQs) pour la gestion des erreurs, mais ne prennent pas en charge les destinations Lambda. Configurez un DLQ pour capturer les enregistrements des appels ayant échoué.

Pour des informations complètes sur la gestion des erreurs de mappage des sources d'événements, consultez la section Mappages des sources d'événements.

Utilisation d'une fonction intermédiaire pour les flux de travail de longue durée

Si votre flux de travail nécessite plus de 15 minutes, utilisez une fonction Lambda standard intermédiaire entre le mappage de la source d'événements et votre fonction durable. La fonction intermédiaire reçoit les événements du mappage de la source d'événements et invoque la fonction durable de manière asynchrone, supprimant ainsi la limite d'exécution de 15 minutes.

Ce modèle dissocie le modèle d'appel synchrone du mappage des sources d'événements du modèle d'exécution à long terme de la fonction durable. Le mappage de la source d'événements invoque la fonction intermédiaire, qui revient rapidement après le démarrage de l'exécution durable. La fonction durable fonctionne ensuite indépendamment aussi longtemps que nécessaire (jusqu'à 1 an).

Architecture

Le modèle de fonction intermédiaire utilise trois composants :

1. Cartographie de la source d'événements : interroge la source de l'événement (Amazon SQS, Kinesis, DynamoDB Streams) et invoque la fonction intermédiaire de manière synchrone avec des lots d'enregistrements.
2. Fonction intermédiaire : fonction Lambda standard qui reçoit les événements du mappage des sources d'événements, valide et transforme les données si nécessaire, et invoque la fonction durable de manière asynchrone. Cette fonction s'exécute rapidement (généralement en moins d'une seconde) et redonne le contrôle au mappage des sources d'événements.
3. Fonction durable : traite l'événement selon une logique complexe en plusieurs étapes qui peut s'exécuter pendant de longues périodes. Invoqué de manière asynchrone, il n'est donc pas limité par la limite de 15 minutes.

Mise en œuvre

La fonction intermédiaire reçoit l'intégralité de l'événement à partir du mappage de la source d'événements et invoque la fonction durable de manière asynchrone. Utilisez le paramètre du nom d'exécution pour vous assurer que l'exécution idempotente démarre, empêchant ainsi le traitement dupliqué si le mappage de la source d'événements tente à nouveau :

TypeScript

```
import { LambdaClient, InvokeCommand } from '@aws-sdk/client-lambda';
import { SQSEvent } from 'aws-lambda';
import { createHash } from 'crypto';

const lambda = new LambdaClient({});

export const handler = async (event: SQSEvent) => {
  // Invoke durable function asynchronously with execution name
  await lambda.send(new InvokeCommand({
```

```
    FunctionName: 'arn:aws:lambda:us-east-1:123456789012:function:my-durable-
function:1',
    InvocationType: 'Event',
    Payload: JSON.stringify({
      executionName: event.Name,
      event: event
    })
  }));

  return { statusCode: 200 };
};
```

Python

```
import boto3
import json
import hashlib

lambda_client = boto3.client('lambda')

def handler(event, context):
    # Invoke durable function asynchronously with execution name
    lambda_client.invoke(
        FunctionName='arn:aws:lambda:us-east-1:123456789012:function:my-durable-
function:1',
        InvocationType='Event',
        Payload=json.dumps({
            'executionName': execution_name,
            'event': event["name"]
        })
    )

    return {'statusCode': 200}
```

Pour garantir l'idempuissance de la fonction intermédiaire elle-même, utilisez Powertools for AWS Lambda afin d'éviter les appels dupliqués de la fonction durable si le mappage de la source d'événements tente à nouveau la fonction intermédiaire.

La fonction durable reçoit la charge utile avec le nom de l'exécution et traite tous les enregistrements selon une logique de longue durée :

TypeScript

```
import { withDurableExecution, DurableContext } from '@aws/durable-execution-sdk-
js';

export const handler = withDurableExecution(
  async (payload: any, context: DurableContext) => {
    const sqsEvent = payload.event;

    // Process each record with complex, multi-step logic
    const results = await context.map(
      sqsEvent.Records,
      async (ctx, record) => {
        const validated = await ctx.step('validate', async () => {
          return validateOrder(JSON.parse(record.body));
        });

        // Wait for external approval (could take hours or days)
        const approval = await ctx.waitForCallback(
          'approval',
          async (callbackId) => {
            await requestApproval(callbackId, validated);
          },
          { timeout: { hours: 48 } }
        );

        // Complete processing
        return await ctx.step('complete', async () => {
          return completeOrder(validated, approval);
        });
      }
    );

    return { statusCode: 200, processed: results.getResults().length };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import durable_execution, DurableContext
import json

@durable_execution
def handler(payload, context: DurableContext):
    sqs_event = payload['event']

    # Process each record with complex, multi-step logic
    def process_record(ctx, record):
        validated = ctx.step(
            lambda _: validate_order(json.loads(record['body'])),
            name='validate'
        )

        # Wait for external approval (could take hours or days)
        approval = ctx.wait_for_callback(
            lambda callback_id: request_approval(callback_id, validated),
            name='approval',
            config=WaitForCallbackConfig(timeout_seconds=172800)  # 48 hours
        )

        # Complete processing
        return ctx.step(
            lambda _: complete_order(validated, approval),
            name='complete'
        )

    results = context.map(sqs_event['Records'], process_record)

    return {'statusCode': 200, 'processed': len(results.get_results())}
```

Considérations clés

Ce modèle supprime la limite d'exécution de 15 minutes en dissociant le mappage de la source d'événements de l'exécution durable. La fonction intermédiaire revient immédiatement après le début de l'exécution durable, ce qui permet au mappage de la source d'événements de poursuivre le traitement. La fonction durable fonctionne ensuite indépendamment aussi longtemps que nécessaire.

La fonction intermédiaire réussit lorsqu'elle invoque la fonction durable, et non lorsque l'exécution durable est terminée. Si l'exécution durable échoue ultérieurement, le mappage de la source d'événements ne réessaiera pas car le lot a déjà été traité avec succès. Implémentez la gestion des erreurs dans la fonction durable et configurez des files d'attente contenant des lettres mortes en cas d'échec d'exécution.

Utilisez le paramètre du nom d'exécution pour vous assurer que l'exécution idempotente démarre. Si le mappage de la source d'événements tente à nouveau la fonction intermédiaire, la fonction durable ne démarrera pas une double exécution car le nom de l'exécution existe déjà.

Sources d'événements prises en charge

Les fonctions durables prennent en charge toutes les sources d'événements Lambda qui utilisent des mappages de sources d'événements :

- Files d'attente Amazon SQS (standard et FIFO)
- Flux Kinesis
- DynamoDB Streams
- Amazon Managed Streaming for Apache Kafka (Amazon MSK)
- Self-managed Apache Kafka
- Amazon MQ (ActiveMQ et RabbitMQ)
- Flux de modifications Amazon DocumentDB

Tous les types de sources d'événements sont soumis à la limite d'exécution durable de 15 minutes lors de l'invocation de fonctions durables.

Réessais pour les fonctions durables de Lambda

Les fonctions durables offrent des fonctionnalités de relance automatique qui permettent à vos applications de résister aux défaillances transitoires. Le SDK gère les nouvelles tentatives à deux niveaux : les tentatives par étapes en cas d'échec de la logique métier et les tentatives du backend en cas de défaillance de l'infrastructure.

Étape 2 : Réessaie

Lorsqu'une exception non détectée se produit au cours d'une étape, le SDK réessaie automatiquement l'étape en fonction de la stratégie de nouvelle tentative configurée. Les tentatives

d'étape sont des opérations ponctuelles qui permettent au SDK de suspendre l'exécution et de la reprendre ultérieurement sans perdre la progression.

Étape 1 : comportement en cas de nouvelle tentative

Le tableau suivant décrit comment le SDK gère les exceptions par étapes :

Scénario	Que se passe-t-il	Impact du mesurage
Exception en cours avec les tentatives de nouvelle tentative restantes	Le SDK crée un point de contrôle pour la nouvelle tentative et suspend la fonction. Lors de l'appel suivant, l'étape réessaie avec le délai d'attente configuré.	1 opération+erreur sur la taille de la charge utile
Exception en cours, aucune nouvelle tentative n'ayant été tentée	L'étape échoue et génère une exception. Si le code de votre gestionnaire ne détecte pas cette exception, l'exécution complète échoue.	1 opération+erreur sur la taille de la charge utile

Lorsqu'une étape doit être réessayée, le SDK vérifie l'état de nouvelle tentative et quitte l'appel Lambda si aucune autre tâche n'est en cours d'exécution. Cela permet au SDK d'implémenter des délais de temporisation sans consommer de ressources informatiques. La fonction reprend automatiquement après la période d'attente.

Configuration des stratégies de nouvelle tentative

Configurez des stratégies de nouvelle tentative pour contrôler la manière dont les étapes gèrent les échecs. Vous pouvez spécifier le nombre maximum de tentatives, les intervalles d'attente et les conditions de réessai.

Retard exponentiel avec un maximum de tentatives :

TypeScript

```
const result = await context.step('call-api', async () => {
  const response = await fetch('https://api.example.com/data');
```

```
  if (!response.ok) throw new Error(`API error: ${response.status}`);
  return await response.json();
}, {
  retryStrategy: (error, attemptCount) => {
    if (attemptCount >= 5) {
      return { shouldRetry: false };
    }
    // Exponential backoff: 2s, 4s, 8s, 16s, 32s (capped at 300s)
    const delay = Math.min(2 * Math.pow(2, attemptCount - 1), 300);
    return { shouldRetry: true, delay: { seconds: delay } };
  }
});
```

Python

```
def retry_strategy(error, attempt_count):
    if attempt_count >= 5:
        return {'should_retry': False}
    # Exponential backoff: 2s, 4s, 8s, 16s, 32s (capped at 300s)
    delay = min(2 * (2 ** (attempt_count - 1)), 300)
    return {'should_retry': True, 'delay': delay}

result = context.step(
    lambda _: call_external_api(),
    name='call-api',
    config=StepConfig(retry_strategy=retry_strategy)
)
```

Annulation à intervalle fixe :

TypeScript

```
const orders = await context.step('query-orders', async () => {
  return await queryDatabase(event.userId);
}, {
  retryStrategy: (error, attemptCount) => {
    if (attemptCount >= 3) {
      return { shouldRetry: false };
```

```
    }
    return { shouldRetry: true, delay: { seconds: 5 } };
  }
});
```

Python

```
def retry_strategy(error, attempt_count):
    if attempt_count >= 3:
        return {'should_retry': False}
    return {'should_retry': True, 'delay': 5}

orders = context.step(
    lambda _: query_database(event['userId']),
    name='query-orders',
    config=StepConfig(retry_strategy=retry_strategy)
)
```

Nouvelle tentative conditionnelle (réessayez uniquement en cas d'erreurs spécifiques) :

TypeScript

```
const result = await context.step('call-rate-limited-api', async () => {
  const response = await fetch('https://api.example.com/data');

  if (response.status === 429) throw new Error('RATE_LIMIT');
  if (response.status === 504) throw new Error('TIMEOUT');
  if (!response.ok) throw new Error(`API_ERROR_${response.status}`);

  return await response.json();
}, {
  retryStrategy: (error, attemptCount) => {
    // Only retry rate limits and timeouts
    const isRetryable = error.message === 'RATE_LIMIT' || error.message ===
 'TIMEOUT';

    if (!isRetryable || attemptCount >= 3) {
      return { shouldRetry: false };
```

```
    }

    // Exponential backoff: 1s, 2s, 4s (capped at 30s)
    const delay = Math.min(Math.pow(2, attemptCount - 1), 30);
    return { shouldRetry: true, delay: { seconds: delay } };
  }
});
```

Python

```
def retry_strategy(error, attempt_count):
    # Only retry rate limits and timeouts
    is_retryable = str(error) in ['RATE_LIMIT', 'TIMEOUT']

    if not is_retryable or attempt_count >= 3:
        return {'should_retry': False}

    # Exponential backoff: 1s, 2s, 4s (capped at 30s)
    delay = min(2 ** (attempt_count - 1), 30)
    return {'should_retry': True, 'delay': delay}

result = context.step(
    lambda _: call_rate_limited_api(),
    name='call-rate-limited-api',
    config=StepConfig(retry_strategy=retry_strategy)
)
```

Désactiver les nouvelles tentatives :

TypeScript

```
const isDuplicate = await context.step('check-duplicate', async () => {
  return await checkIfOrderExists(event.orderId);
}, {
  retryStrategy: () => ({ shouldRetry: false })
});
```

Python

```
is_duplicate = context.step(
    lambda _: check_if_order_exists(event['orderId']),
    name='check-duplicate',
    config=StepConfig(
        retry_strategy=lambda error, attempt: {'should_retry': False}
    )
)
```

Lorsque la stratégie de nouvelle tentative revient`shouldRetry: false`, l'étape échoue immédiatement sans nouvelle tentative. Utilisez-le pour les opérations qui ne doivent pas être renouvelées, telles que les contrôles d'idempuissance ou les opérations présentant des effets secondaires qui ne peuvent pas être répétés en toute sécurité.

Exceptions en dehors des étapes

Lorsqu'une exception non détectée se produit dans le code de votre gestionnaire mais en dehors de toute étape, le SDK marque l'exécution comme ayant échoué. Cela garantit que les erreurs dans la logique de votre application sont correctement capturées et signalées.

Scénario	Que se passe-t-il	Impact du mesurage
Exception dans le code du gestionnaire en dehors de toute étape	Le SDK marque l'exécuti on comme ayant échoué et renvoie l'erreur. L'exception n'est pas automatiquement réessayée.	Erreur concernant la taille de la charge utile

Pour activer la nouvelle tentative automatique pour le code sujet aux erreurs, associez-le à une étape avec une stratégie de nouvelle tentative. Les étapes fournissent une nouvelle tentative automatique avec une interruption configurable, tandis que le code en dehors des étapes échoue immédiatement.

Rétentatives du backend

Les nouvelles tentatives du backend ont lieu lorsque Lambda rencontre des défaillances d'infrastructure, des erreurs d'exécution ou lorsque le SDK ne peut pas communiquer avec le service d'exécution durable. Lambda réessaie automatiquement ces défaillances pour garantir que vos fonctions durables puissent se rétablir après des problèmes d'infrastructure transitoires.

Scénarios de nouvelle tentative du backend

Lambda réessaie automatiquement votre fonction lorsqu'elle rencontre les scénarios suivants :

- Erreurs de service internes : lorsque Lambda ou le service d'exécution durable renvoie une erreur 5xx, indiquant un problème de service temporaire.
- Limitation : lorsque votre fonction est limitée en raison de limites de simultanéité ou de quotas de service.
- Délais d'expiration : lorsque le SDK ne parvient pas à atteindre le service d'exécution durable dans le délai imparti.
- Échec de l'initialisation du sandbox : lorsque Lambda ne parvient pas à initialiser l'environnement d'exécution.
- Erreurs d'exécution : lorsque le moteur d'exécution Lambda rencontre des erreurs extérieures au code de votre fonction, telles que out-of-memory des erreurs ou des pannes de processus.
- Erreurs de jeton de point de contrôle non valides : lorsque le jeton de point de contrôle n'est plus valide, généralement en raison de changements d'état côté service.

Le tableau suivant décrit comment le SDK gère ces scénarios :

Scénario	Que se passe-t-il	Impact du mesurage
Erreur d'exécution en dehors du gestionnaire durable (OOM, délai d'attente, crash)	Lambda réessaie automatiquement l'invocation. Le SDK se rejoue depuis le dernier point de contrôle, en sautant les étapes terminées.	Taille de la charge utile d'erreur + 1 opération par nouvelle tentative
Erreur de service (5xx) ou délai d'expiration lors	Lambda réessaie automatiquement l'invocation. Le SDK se rejoue depuis le dernier point de contrôle.	Taille de la charge utile d'erreur + 1 opération par nouvelle tentative

Scénario	Que se passe-t-il	Impact du mesurage
de l'appel/`CheckpointDurableExecution` `GetDurableExecutionState` APIs		
Throttling (429) ou jeton de point de contrôle non valide lors de l'appel/`CheckpointDurableExecution` `GetDurableExecutionState` APIs	Lambda réessaie automatiquement l'invocation avec un retard exponentiel. Le SDK se rejoue depuis le dernier point de contrôle.	Taille de la charge utile d'erreur + 1 opération par nouvelle tentative
Erreur client (4xx, sauf 429 et jeton non valide) lorsque/`CheckpointDurableExecution` `GetDurableExecutionState` APIs	Le SDK marque l'exécution comme ayant échoué. Aucune nouvelle tentative automatique ne se produit car l'erreur indique un problème permanent.	Erreur concernant la taille de la charge utile

Les nouvelles tentatives du backend utilisent un retard exponentiel et se poursuivent jusqu'à ce que la fonction réussisse ou que le délai d'exécution soit atteint. Pendant la rediffusion, le SDK ignore les points de contrôle terminés et poursuit l'exécution depuis la dernière opération réussie, garantissant ainsi que votre fonction ne réexécute pas le travail terminé.

Réessayez les meilleures pratiques

Suivez les meilleures pratiques suivantes lors de la configuration des stratégies de nouvelle tentative :

- Configurez des stratégies de relance explicites : ne vous fiez pas au comportement de nouvelle tentative par défaut en production. Configurez des stratégies de nouvelles tentatives explicites avec un nombre maximum de tentatives et des intervalles d'attente adaptés à votre cas d'utilisation.

- Utiliser des tentatives conditionnelles : implémentez une `shouldRetry` logique permettant de réessayer uniquement les erreurs transitoires (limites de débit, délais d'attente) et d'échouer rapidement en cas d'erreur permanente (échec de validation, introuvable).
- Définissez le nombre maximum de tentatives approprié : équilibre entre résilience et temps d'exécution. Un trop grand nombre de tentatives peut retarder la détection des échecs, tandis qu'un nombre insuffisant peut entraîner des échecs inutiles.
- Utilisez le ralentissement exponentiel : le ralentissement exponentiel réduit la charge sur les services en aval et augmente les chances de reprise après des défaillances transitoires.
- Enveloppez le code sujet aux erreurs par étapes : le code en dehors des étapes ne peut pas être réessayé automatiquement. Intégrez les appels d'API externes, les requêtes de base de données et les autres opérations susceptibles d'entraîner des erreurs par étapes grâce à des stratégies de nouvelle tentative.
- Surveillez les statistiques relatives aux nouvelles tentatives : suivez les opérations de nouvelle tentative par étapes et les échecs d'exécution sur Amazon CloudWatch afin d'identifier les modèles et d'optimiser les stratégies de nouvelles tentatives.

Idempotence

Les fonctions durables fournissent une idempuissance intégrée pour les démarrages d'exécution grâce aux noms d'exécution. Lorsque vous fournissez un nom d'exécution, Lambda l'utilise pour empêcher les doubles exécutions et permettre de nouvelles tentatives d'appel en toute sécurité. Les étapes ont une sémantique at-least-once d'exécution par défaut : lors de la réexécution, le SDK renvoie des résultats précis sans réexécuter les étapes terminées, mais votre logique métier doit être idempotente pour gérer d'éventuelles nouvelles tentatives avant la fin.

> **Note**
>
> Les mappages de sources d'événements Lambda (ESM) ne prennent pas en charge l'idempotencie au lancement. Par conséquent, chaque invocation (y compris les nouvelles tentatives) lance une nouvelle exécution durable. Pour garantir une exécution idempotente avec des mappages de sources d'événements, implémentez la logique d'idempotentie dans votre code de fonction, par exemple avec Powertools for, ou AWS Lambda utilisez une fonction Lambda normale comme proxy (répartiteur) pour appeler une fonction durable avec une clé d'idempotentie (paramètre du nom d'exécution).

Noms d'exécution

Vous pouvez fournir un nom d'exécution lorsque vous invoquez une fonction durable. Le nom d'exécution agit comme une clé d'impuissance, vous permettant de réessayer les demandes d'invocation en toute sécurité sans créer de doublons d'exécutions. Si vous ne fournissez pas de nom, Lambda génère automatiquement un identifiant d'exécution unique.

Les noms d'exécution doivent être uniques au sein de votre compte et de votre région. Lorsque vous invoquez une fonction dont le nom d'exécution existe déjà, le comportement de Lambda dépend de l'état de l'exécution existante et de la correspondance de la charge utile.

Comportement d'impuissance

Le tableau suivant décrit comment Lambda gère les demandes d'invocation selon que vous fournissez un nom d'exécution, l'état d'exécution existant et si la charge utile correspond :

Scénario	Nom fourni ?	État d'exécution existant	Charge utile identique ?	Comportement
1	Non	N/A	N/A	Nouvelle exécution démarrée : Lambda génère un ID d'exécution unique et lance une nouvelle exécution
2	Oui	N'a jamais existé ou la rétention a expiré	N/A	Nouvelle exécution démarrée : Lambda lance une nouvelle exécution avec le nom fourni
3	Oui	En cours d'exécution	Oui	Démarrage idempotent : Lambda renvoie les informations d'exécution existantes sans créer de doublon. Pour les invocations synchrones, cela agit

Scénario	Nom fourni ?	État d'exécution existant	Charge utile identique ?	Comportement
				comme un rattachement à l'exécution en cours
4	Oui	En cours d’exécution	Non	Erreur : Lambda renvoie une `DurableExecutionAlreadyExists` erreur car une exécution portant ce nom est déjà en cours d'exécution avec une charge utile différente
5	Oui	Fermé (réussite, échec, arrêt ou expiration du délai imparti)	Oui	Démarrage idempotent : Lambda renvoie les informations d'exécution existantes sans démarrer une nouvelle exécution. Le résultat de l'exécution fermée est renvoyé
6	Oui	Fermé (réussite, échec, arrêt ou expiration du délai imparti)	Non	Erreur : Lambda renvoie une `DurableExecutionAlreadyExists` erreur car une exécution portant ce nom est déjà terminée avec une charge utile différente

Remarque

Les scénarios 3 et 5 illustrent un comportement idempotent dans lequel Lambda gère en toute sécurité les demandes d'invocation dupliquées en renvoyant les informations d'exécution existantes au lieu de créer des doublons.

Étape 1 : Idempuissance

Les étapes ont une sémantique at-least-once d'exécution par défaut. Lorsque votre fonction est rejouée après une attente, un rappel ou un échec, le SDK compare chaque étape au journal des points de contrôle. Pour les étapes déjà terminées, le SDK renvoie le résultat du point de contrôle sans réexécuter la logique des étapes. Toutefois, si une étape échoue ou si la fonction est interrompue avant la fin de l'étape, l'étape peut être exécutée plusieurs fois.

Votre logique métier articulée en étapes doit être idempotente pour gérer d'éventuelles nouvelles tentatives. Utilisez des clés d'idempuissance pour garantir que les opérations telles que les paiements ou les écritures de base de données ne s'exécutent qu'une seule fois, même si l'étape est réessayée.

Exemple : utilisation des clés d'idempuissance par étapes

TypeScript

```
import { withDurableExecution, DurableContext } from '@aws/durable-execution-sdk-
js';
import { randomUUID } from 'crypto';

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    // Generate idempotency key once
    const idempotencyKey = await context.step('generate-key', async () => {
      return randomUUID();
    });

    // Use idempotency key in payment API to prevent duplicate charges
    const payment = await context.step('process-payment', async () => {
      return paymentAPI.charge({
        amount: event.amount,
        idempotencyKey: idempotencyKey
      });
    });
```

```
    return { statusCode: 200, payment };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import durable_execution, DurableContext
import uuid

@durable_execution
def handler(event, context: DurableContext):
    # Generate idempotency key once
    idempotency_key = context.step(
        lambda _: str(uuid.uuid4()),
        name='generate-key'
    )

    # Use idempotency key in payment API to prevent duplicate charges
    payment = context.step(
        lambda _: payment_api.charge(
            amount=event['amount'],
            idempotency_key=idempotency_key
        ),
        name='process-payment'
    )

    return {'statusCode': 200, 'payment': payment}
```

Vous pouvez configurer les étapes pour utiliser la sémantique at-most-once d'exécution en définissant le mode d'exécution sur. `AT_MOST_ONCE_PER_RETRY` Cela garantit que l'étape s'exécute au maximum une fois par nouvelle tentative, mais peut ne pas s'exécuter du tout si la fonction est interrompue avant la fin de l'étape.

Le SDK applique la rediffusion déterministe en vérifiant que les noms et l'ordre des étapes correspondent au journal des points de contrôle lors de la rediffusion. Si votre code tente d'exécuter des étapes dans un ordre différent ou avec des noms différents, le SDK lance un`NonDeterministicExecutionError`.

Comment fonctionne le replay une fois les étapes terminées :

1. Premier appel : la fonction exécute l'étape A, crée un point de contrôle, puis attend
2. Deuxième appel (après attente) : la fonction est rejouée depuis le début, l'étape A renvoie instantanément le résultat vérifié sans la réexécuter, puis passe à l'étape B
3. Troisième appel (après une nouvelle attente) : les fonctions sont rejouées depuis le début, les étapes A et B renvoient instantanément les résultats vérifiés, puis passent à l'étape C

Ce mécanisme de réexécution garantit que les étapes terminées ne sont pas réexécutées, mais votre logique métier doit toujours être idempotente pour gérer les nouvelles tentatives avant leur fin.

Tester les fonctions durables de Lambda

AWS fournit des tests dédiés SDKs pour les fonctions durables qui vous permettent d'exécuter et d'inspecter les exécutions à la fois localement et dans le cloud. Installez le SDK de test pour votre langue :

TypeScript

```
npm install --save-dev @aws/aws-durable-execution-sdk-js-testing
```

Pour obtenir une documentation complète et des exemples, consultez le SDK de TypeScript test sur GitHub.

Python

```
pip install aws-durable-execution-sdk-python-testing
```

Pour une documentation complète et des exemples, consultez le SDK de test Python sur GitHub.

Le SDK de test propose deux modes de test : les tests locaux pour les tests unitaires rapides et les tests dans le cloud pour les tests d'intégration sur les fonctions déployées.

Tests locaux

Les tests locaux exécutent vos fonctions durables dans votre environnement de développement sans nécessiter de ressources déployées. Le lanceur de tests exécute directement votre code de fonction et capture toutes les opérations à des fins d'inspection.

Utilisez les tests locaux pour les tests unitaires, le développement piloté par les tests et CI/CD les pipelines. Les tests sont exécutés localement sans latence réseau ni coûts supplémentaires.

Exemple de test :

TypeScript

```
import { withDurableExecution } from '@aws/aws-durable-execution-sdk-js';
import { DurableFunctionTestRunner } from '@aws/aws-durable-execution-sdk-js-
testing';

const handler = withDurableExecution(async (event, context) => {
  const result = await context.step('calculate', async () => {
    return event.a + event.b;
  });
  return result;
});

test('addition works correctly', async () => {
  const runner = new DurableFunctionTestRunner({ handler });
  const result = await runner.run({ a: 5, b: 3 });

  expect(result.status).toBe('SUCCEEDED');
  expect(result.result).toBe(8);

  const step = result.getStep('calculate');
  expect(step.result).toBe(8);
});
```

Python

```
from aws_durable_execution_sdk_python import durable_execution, DurableContext
from aws_durable_execution_sdk_python_testing import DurableFunctionTestRunner
from aws_durable_execution_sdk_python.execution import InvocationStatus
```

```
@durable_execution
def handler(event: dict, context: DurableContext) -> int:
    result = context.step(lambda _: event["a"] + event["b"], name="calculate")
    return result

def test_addition():
    runner = DurableFunctionTestRunner(handler=handler)
    with runner:
        result = runner.run(input={"a": 5, "b": 3}, timeout=10)

    assert result.status is InvocationStatus.SUCCEEDED
    assert result.result == 8

    step = result.get_step("calculate")
    assert step.result == 8
```

Le lanceur de tests capture l'état d'exécution, y compris le résultat final, les résultats des étapes individuelles, les opérations d'attente, les rappels et les éventuelles erreurs. Vous pouvez inspecter les opérations par leur nom ou répéter toutes les opérations pour vérifier le comportement d'exécution.

Magasins d'exécution

Le SDK de test utilise des magasins d'exécution pour conserver les données d'exécution des tests. Par défaut, les tests utilisent un magasin en mémoire qui est rapide et ne nécessite aucun nettoyage. Pour le débogage ou l'analyse de l'historique des exécutions, vous pouvez utiliser un magasin de système de fichiers qui enregistre les exécutions sous forme de fichiers JSON.

Stockage en mémoire (par défaut) :

Le stockage en mémoire conserve les données d'exécution en mémoire pendant les tests. Les données sont perdues à la fin des tests, ce qui les rend idéales pour les tests unitaires standard et les CI/CD pipelines où vous n'avez pas besoin d'inspecter les exécutions une fois les tests terminés.

Stockage du système de fichiers :

Le magasin du système de fichiers conserve les données d'exécution sur le disque sous forme de fichiers JSON. Chaque exécution est enregistrée dans un fichier distinct, ce qui permet de consulter facilement l'historique des exécutions une fois les tests terminés. Utilisez le magasin du système de

fichiers pour corriger les échecs de tests complexes ou pour analyser les modèles d'exécution au fil du temps.

Configurez le magasin à l'aide de variables d'environnement :

```
# Use filesystem store
export AWS_DEX_STORE_TYPE=filesystem
export AWS_DEX_STORE_PATH=./test-executions

# Run tests
pytest tests/
```

Les fichiers d'exécution sont stockés avec des noms nettoyés et contiennent l'état d'exécution complet, y compris les opérations, les points de contrôle et les résultats. Le magasin du système de fichiers crée automatiquement le répertoire de stockage s'il n'existe pas.

Test cloud

Les tests dans le cloud invoquent les fonctions durables déployées AWS et récupèrent leur historique d'exécution à l'aide de l'API Lambda. Utilisez les tests dans le cloud pour vérifier le comportement dans des environnements de production dotés de AWS services et de configurations réels.

Les tests dans le cloud nécessitent une fonction déployée et des AWS informations d'identification avec les autorisations nécessaires pour appeler des fonctions et lire l'historique d'exécution :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction",
                "lambda:GetDurableExecution",
                "lambda:GetDurableExecutionHistory"
            ],
            "Resource": "arn:aws:lambda:region:account-id:function:function-name"
        }
    ]
}
```

Exemple de test dans le cloud :

TypeScript

```
import { DurableFunctionCloudTestRunner } from '@aws/aws-durable-execution-sdk-js-
testing';

test('deployed function processes orders', async () => {
  const runner = new DurableFunctionCloudTestRunner({
    functionName: 'order-processor',
    region: 'us-east-1'
  });

  const result = await runner.run({ orderId: 'order-123' });

  expect(result.status).toBe('SUCCEEDED');
  expect(result.result.status).toBe('completed');
});
```

Python

```
from aws_durable_execution_sdk_python_testing import (
    DurableFunctionCloudTestRunner,
    DurableFunctionCloudTestRunnerConfig
)

def test_deployed_function():
    config = DurableFunctionCloudTestRunnerConfig(
        function_name="order-processor",
        region="us-east-1"
    )
    runner = DurableFunctionCloudTestRunner(config=config)

    result = runner.run(input={"orderId": "order-123"})

    assert result.status is InvocationStatus.SUCCEEDED
    assert result.result["status"] == "completed"
```

Les tests dans le cloud invoquent la fonction réellement déployée et en extraient l'historique d'exécution AWS. Cela vous permet de vérifier l'intégration avec d'autres AWS services, de valider les caractéristiques de performance et de tester avec des données et des configurations similaires à celles de la production.

Que faut-il tester ?

Testez les fonctions durables en vérifiant les résultats d'exécution, le comportement des opérations et la gestion des erreurs. Concentrez-vous sur l'exactitude de la logique métier plutôt que sur les détails de mise en œuvre.

Vérifiez les résultats de l'exécution : vérifiez que les fonctions renvoient les valeurs attendues pour des entrées données. Testez à la fois les exécutions réussies et les cas d'erreur pour vous assurer que les fonctions gèrent correctement les entrées non valides.

Inspectez l'exécution des opérations : vérifiez que les étapes, les attentes et les rappels s'exécutent comme prévu. Vérifiez les résultats des étapes pour vous assurer que les opérations intermédiaires produisent des valeurs correctes. Vérifiez que les opérations d'attente sont configurées avec des délais d'expiration appropriés et que les rappels sont créés avec les paramètres appropriés.

Gestion des erreurs de test : vérifiez que les fonctions échouent correctement avec des messages d'erreur descriptifs en cas d'entrée non valide. Testez le comportement des nouvelles tentatives en simulant des défaillances transitoires et en vérifiant que les opérations recommencent correctement. Vérifiez que les échecs permanents ne déclenchent pas de nouvelles tentatives inutiles.

Valider les flux de travail : pour les flux de travail en plusieurs étapes, vérifiez que les opérations s'exécutent dans le bon ordre. Testez le branchement conditionnel pour vous assurer que les différents chemins d'exécution fonctionnent correctement. Validez les opérations parallèles exécutées simultanément et produisez les résultats attendus.

Les référentiels de documentation du SDK contiennent de nombreux exemples de modèles de test, notamment des flux de travail en plusieurs étapes, des scénarios d'erreur, la gestion des délais d'attente et des modèles de sondage.

Stratégie de test

Utilisez des tests locaux pour les tests unitaires pendant le développement et dans les CI/CD pipelines. Les tests locaux s'exécutent rapidement, ne nécessitent pas AWS d'informations d'identification et fournissent des informations immédiates sur les modifications du code. Rédigez des tests locaux pour vérifier la logique métier, la gestion des erreurs et le comportement des opérations.

Utilisez les tests dans le cloud pour les tests d'intégration avant le déploiement en production. Les tests dans le cloud vérifient le comportement à l'aide de AWS services et de configurations réels, valident les caractéristiques de performance et testent end-to-end les flux de travail. Exécutez des tests cloud dans des environnements de test afin de détecter les problèmes d'intégration avant qu'ils ne soient mis en production.

Simulez les dépendances externes dans les tests locaux pour isoler la logique des fonctions et accélérer les tests. Utilisez des tests dans le cloud pour vérifier l'intégration réelle avec des services externes tels que des bases de données et d'autres AWS services. APIs

Rédigez des tests ciblés qui vérifient un comportement spécifique. Utilisez des noms de test descriptifs qui expliquent ce qui est testé. Regroupez les tests connexes et utilisez des appareils de test pour un code de configuration commun. Simplifiez les tests et évitez les logiques de test complexes difficiles à comprendre.

Défaillances de débogage

Lorsque les tests échouent, inspectez le résultat de l'exécution pour comprendre ce qui s'est mal passé. Vérifiez l'état d'exécution pour voir si la fonction a réussi, a échoué ou a expiré. Lisez les messages d'erreur pour comprendre la cause de l'échec.

Inspectez les résultats de chaque opération pour trouver les domaines dans lesquels le comportement ne correspondait pas aux attentes. Vérifiez les résultats des étapes pour voir quelles valeurs ont été produites. Vérifiez l'ordre des opérations pour confirmer que les opérations sont exécutées dans l'ordre attendu. Comptez les opérations pour vous assurer que le bon nombre d'étapes, d'attentes et de rappels a été créé.

Les problèmes courants incluent le code non déterministe qui produit des résultats différents lors de la réexécution, l'état partagé par le biais de variables globales qui s'interrompent pendant la rediffusion et les opérations manquantes en raison d'erreurs de logique conditionnelle. Utilisez les débogueurs et la journalisation standard pour parcourir le code de fonction et suivre le flux d'exécution.

Pour les tests dans le cloud, consultez l'historique des exécutions dans CloudWatch les journaux pour consulter les journaux d'opérations détaillés. Utilisez le suivi pour suivre le flux d'exécution entre les services et identifier les goulots d'étranglement.

Surveillance des fonctions durables

Vous pouvez surveiller vos fonctions durables à l'aide de CloudWatch métriques, de CloudWatch journaux et de traçage. Les fonctions durables pouvant s'exécuter pendant de longues périodes et couvrir plusieurs invocations de fonctions, leur surveillance nécessite de comprendre leurs modèles d'exécution uniques, notamment les points de contrôle, les transitions d'état et le comportement de rediffusion.

CloudWatch métriques

Lambda publie automatiquement les métriques sans CloudWatch frais supplémentaires. Les fonctions durables fournissent des mesures supplémentaires au-delà des métriques Lambda standard pour vous aider à surveiller les flux de travail de longue durée, la gestion des états et l'utilisation des ressources.

Mesures d'exécution durables

Lambda émet les métriques suivantes pour des exécutions durables :

Métrique	Description
`ApproximateRunningDurableExecutions`	Nombre d'exécutions durables à l'état RUNNING
`ApproximateRunningDurableExecutionsUtilization`	Pourcentage du quota maximal d'exécutions durables de votre compte actuellement utilisé
`DurableExecutionDuration`	Temps écoulé, en millisecondes, pendant lequel une exécution durable est restée à l'état RUNNING
`DurableExecutionStarted`	Nombre d'exécutions durables qui ont débuté
`DurableExecutionStopped`	Nombre d'exécutions durables interrompues à l'aide de l'StopDurableExecution API

Métrique	Description
`DurableExecutionSucceeded`	Nombre d'exécutions durables achevées avec succès
`DurableExecutionFailed`	Nombre d'exécutions durables qui se sont soldées par un échec
`DurableExecutionTimedOut`	Nombre d'exécutions durables ayant dépassé le délai d'exécution configuré
`DurableExecutionOperations`	Nombre cumulé d'opérations effectuées dans le cadre d'une exécution durable (maximum : 3 000)
`DurableExecutionStorageWrittenBytes`	Quantité cumulée de données en octets persistée par une exécution durable (max. : 100 Mo)

CloudWatch métriques

Lambda émet des métriques d'invocation, de performance et de simultanéité standard pour des fonctions durables. Étant donné qu'une exécution durable peut s'étendre sur plusieurs invocations de fonctions au fur et à mesure qu'elle progresse entre les points de contrôle et les rediffusions, ces métriques se comportent différemment de celles des fonctions standard :

- Invocations : compte chaque appel de fonction, y compris les rediffusions. Une seule exécution durable peut générer plusieurs points de données d'invocation.
- Durée : mesure chaque invocation de fonction séparément. `DurableExecutionDuration`À utiliser pour le temps total nécessaire à une seule exécution durable.
- Erreurs : suit les échecs d'invocation des fonctions. À utiliser `DurableExecutionFailed` pour les échecs au niveau de l'exécution.

Pour une liste complète des métriques Lambda standard, voir Types de métriques pour les fonctions Lambda.

Création d' CloudWatch alarmes

Créez des CloudWatch alarmes pour vous avertir lorsque les mesures dépassent les seuils. Les alarmes les plus courantes incluent :

- `ApproximateRunningDurableExecutionsUtilization`dépasse 80 % de votre quota
- `DurableExecutionFailed`augmentations supérieures à un seuil
- `DurableExecutionTimedOut`indique que le délai d'exécution est expiré
- `DurableExecutionStorageWrittenBytes`approche les limites de stockage

Pour plus d'informations, consultez la section Utilisation des CloudWatch alarmes. .

EventBridge événements

Lambda publie les événements de modification du statut d'exécution durable sur. EventBridge Vous pouvez utiliser ces événements pour déclencher des flux de travail, envoyer des notifications ou suivre les modifications du cycle de vie d'exécution de vos fonctions durables.

Événements de changement de statut d'exécution durables

Lambda émet un événement EventBridge chaque fois qu'une exécution durable change de statut. Ces événements présentent les caractéristiques suivantes :

- Source : `aws.lambda`
- Type de détail : `Durable Execution Status Change`

Les événements de changement de statut sont publiés pour les états d'exécution suivants :

- `RUNNING`- L'exécution a commencé
- `SUCCEEDED`- Exécution terminée avec succès
- `STOPPED`- L'exécution s'est arrêtée à l'aide de l' StopDurableExecution API
- `FAILED`- L'exécution a échoué avec une erreur
- `TIMED_OUT`- L'exécution a dépassé le délai d'expiration configuré

L'exemple suivant montre un événement de changement de statut d'exécution durable :

```
{
  "version": "0",
  "id": "d019b03c-a8a3-9d58-85de-241e96206538",
  "detail-type": "Durable Execution Status Change",
  "source": "aws.lambda",
```

```
  "account": "123456789012",
  "time": "2025-11-20T13:08:22Z",
  "region": "us-east-1",
  "resources": [],
  "detail": {
    "durableExecutionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function:
$LATEST/durable-execution/090c4189-b18b-4296-9d0c-cfd01dc3a122/9f7d84c9-ea3d-3ffc-
b3e5-5ec51c34ffc9",
    "durableExecutionName": "order-123",
    "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function:2",
    "status": "RUNNING",
    "startTimestamp": "2025-11-20T13:08:22.345Z"
  }
}
```

Pour les états du terminal (`SUCCEEDED`,`STOPPED`,`FAILED`,,`TIMED_OUT`), l'événement inclut un `endTimestamp` champ indiquant quand l'exécution est terminée.

Création de EventBridge règles

Créez des règles pour acheminer les événements de changement de statut d'exécution durables vers des cibles telles qu'Amazon Simple Notification Service, Amazon Simple Queue Service ou d'autres fonctions Lambda.

L'exemple suivant crée une règle qui correspond à toutes les modifications du statut d'exécution durable :

```
{
  "source": ["aws.lambda"],
  "detail-type": ["Durable Execution Status Change"]
}
```

L'exemple suivant crée une règle qui ne correspond qu'aux exécutions ayant échoué :

```
{
  "source": ["aws.lambda"],
  "detail-type": ["Durable Execution Status Change"],
  "detail": {
```

```
    "status": ["FAILED"]
  }
}
```

L'exemple suivant crée une règle qui correspond aux changements de statut pour une fonction spécifique :

```
{
  "source": ["aws.lambda"],
  "detail-type": ["Durable Execution Status Change"],
  "detail": {
    "functionArn": [{
      "prefix": "arn:aws:lambda:us-east-1:123456789012:function:my-function"
    }]
  }
}
```

Pour plus d'informations sur la création de règles, consultez les EventBridge didacticiels Amazon dans le guide de EventBridge l'utilisateur.

AWS X-Ray traçage

Vous pouvez activer le traçage X-Ray sur vos fonctions durables. Lambda transmet l'en-tête de trace X-Ray à l'exécution durable, ce qui vous permet de suivre les demandes dans l'ensemble de votre flux de travail.

Pour activer le traçage X-Ray à l'aide de la console Lambda, choisissez votre fonction, puis choisissez Configuration, Monitoring and Operations tools, puis activez Active Tracing sous X-Ray.

Pour activer le traçage X-Ray à l'aide de AWS CLI :

```
aws lambda update-function-configuration \
    --function-name my-durable-function \
    --tracing-config Mode=Active
```

Pour activer le AWS X-Ray suivi à l'aide de AWS SAM :

```
Resources:
  MyDurableFunction:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      DurableConfig:
        ExecutionTimeout: 3600
```

Pour plus d'informations sur X-Ray, consultez le manuel du AWS X-Ray développeur.

Meilleures pratiques pour les fonctions durables de Lambda

Les fonctions durables utilisent un modèle d'exécution basé sur le replay qui nécessite des modèles différents de ceux des fonctions Lambda traditionnelles. Suivez ces meilleures pratiques pour créer des flux de travail fiables et rentables.

Écrire du code déterministe

Pendant la réexécution, votre fonction s'exécute depuis le début et doit suivre le même chemin d'exécution que l'exécution initiale. Le code en dehors des opérations durables doit être déterministe et produire les mêmes résultats avec les mêmes entrées.

Répartissez les opérations non déterministes par étapes :

- Génération de nombres aléatoires et UUIDs
- Heure actuelle ou horodatage
- Appels d'API externes et requêtes de base de données
- Opérations du système de fichiers

TypeScript

```
import { withDurableExecution, DurableContext } from '@aws/durable-execution-sdk-
js';
import { randomUUID } from 'crypto';
```

```
export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    // Generate transaction ID inside a step
    const transactionId = await context.step('generate-transaction-id', async () =>
 {
      return randomUUID();
    });

    // Use the same ID throughout execution, even during replay
    const payment = await context.step('process-payment', async () => {
      return processPayment(event.amount, transactionId);
    });

    return { statusCode: 200, transactionId, payment };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import durable_execution, DurableContext
import uuid

@durable_execution
def handler(event, context: DurableContext):
    # Generate transaction ID inside a step
    transaction_id = context.step(
        lambda _: str(uuid.uuid4()),
        name='generate-transaction-id'
    )

    # Use the same ID throughout execution, even during replay
    payment = context.step(
        lambda _: process_payment(event['amount'], transaction_id),
        name='process-payment'
    )

    return {'statusCode': 200, 'transactionId': transaction_id, 'payment': payment}
```

Important

N'utilisez pas de variables globales ou de fermetures pour partager l'état entre les étapes. Transmettez les données via les valeurs de retour. L'état global se brise pendant la rediffusion car les étapes renvoient les résultats mis en cache mais les variables globales sont réinitialisées.

Évitez les mutations de fermeture : les variables capturées lors des fermetures peuvent perdre des mutations pendant la rediffusion. Les étapes renvoient les résultats mis en cache, mais les mises à jour des variables en dehors de l'étape ne sont pas reproduites.

TypeScript

```
// # WRONG: Mutations lost on replay
export const handler = withDurableExecution(async (event, context) => {
  let total = 0;

  for (const item of items) {
    await context.step(async () => {
      total += item.price; // ## Mutation lost on replay!
      return saveItem(item);
    });
  }

  return { total }; // Inconsistent value!
});

// # CORRECT: Accumulate with return values
export const handler = withDurableExecution(async (event, context) => {
  let total = 0;

  for (const item of items) {
    total = await context.step(async () => {
      const newTotal = total + item.price;
      await saveItem(item);
      return newTotal; // Return updated value
    });
  }

  return { total }; // Consistent!
});

// # EVEN BETTER: Use map for parallel processing
export const handler = withDurableExecution(async (event, context) => {
  const results = await context.map(
```

```
    items,
    async (ctx, item) => {
      await ctx.step(async () => saveItem(item));
      return item.price;
    }
  );

  const total = results.getResults().reduce((sum, price) => sum + price, 0);
  return { total };
});
```

Python

```
# # WRONG: Mutations lost on replay
@durable_execution
def handler(event, context: DurableContext):
    total = 0

    for item in items:
        context.step(
            lambda _: save_item_and_mutate(item, total),  # ## Mutation lost on
 replay!
            name=f'save-item-{item["id"]}'
        )

    return {'total': total}  # Inconsistent value!

# # CORRECT: Accumulate with return values
@durable_execution
def handler(event, context: DurableContext):
    total = 0

    for item in items:
        total = context.step(
            lambda _: save_item_and_return_total(item, total),
            name=f'save-item-{item["id"]}'
        )

    return {'total': total}  # Consistent!

# # EVEN BETTER: Use map for parallel processing
```

```
@durable_execution
def handler(event, context: DurableContext):
    def process_item(ctx, item):
        ctx.step(lambda _: save_item(item))
        return item['price']

    results = context.map(items, process_item)
    total = sum(results.get_results())

    return {'total': total}
```

Conception axée sur l'idempuissance

Les opérations peuvent être exécutées plusieurs fois en raison de nouvelles tentatives ou de rediffusions. Les opérations non idempotentes entraînent des effets secondaires dupliqués, comme le fait de facturer deux fois les clients ou d'envoyer plusieurs e-mails.

Utilisez des jetons d'idempuissance : générez des jetons en quelques étapes et incluez-les dans les appels d'API externes pour éviter les opérations dupliquées.

TypeScript

```
import { withDurableExecution, DurableContext } from '@aws/durable-execution-sdk-
js';

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    // Generate idempotency token once
    const idempotencyToken = await context.step('generate-idempotency-token', async
 () => {
      return crypto.randomUUID();
    });

    // Use token to prevent duplicate charges
    const charge = await context.step('charge-payment', async () => {
      return paymentService.charge({
        amount: event.amount,
        cardToken: event.cardToken,
        idempotencyKey: idempotencyToken
      });
```

```
    });

    return { statusCode: 200, charge };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import durable_execution, DurableContext
import uuid

@durable_execution
def handler(event, context: DurableContext):
    # Generate idempotency token once
    idempotency_token = context.step(
        lambda _: str(uuid.uuid4()),
        name='generate-idempotency-token'
    )

    # Use token to prevent duplicate charges
    def charge_payment(_):
        return payment_service.charge(
            amount=event['amount'],
            card_token=event['cardToken'],
            idempotency_key=idempotency_token
        )

    charge = context.step(charge_payment, name='charge-payment')

    return {'statusCode': 200, 'charge': charge}
```

Utilisez la at-most-once sémantique : pour les opérations critiques qui ne doivent jamais être dupliquées (transactions financières, déductions d'inventaire), configurez at-most-once le mode d'exécution.

TypeScript

```
// Critical operation that must not duplicate
```

```
await context.step('deduct-inventory', async () => {
  return inventoryService.deduct(event.productId, event.quantity);
}, {
  executionMode: 'AT_MOST_ONCE_PER_RETRY'
});
```

Python

```
# Critical operation that must not duplicate
context.step(
    lambda _: inventory_service.deduct(event['productId'], event['quantity']),
    name='deduct-inventory',
    config=StepConfig(execution_mode='AT_MOST_ONCE_PER_RETRY')
)
```

Identité de la base de données : utilisez check-before-write des modèles, des mises à jour conditionnelles ou des opérations de modification pour éviter les doublons d'enregistrements.

Gérez efficacement l'état

Chaque point de contrôle enregistre l'état dans un stockage permanent. Les objets d'état volumineux augmentent les coûts, ralentissent le pointage des points de contrôle et ont un impact sur les performances. Stockez uniquement les données essentielles de coordination du flux de travail.

Maintenez l'état au minimum :

- Magasin IDs et références, pas des objets complets
- Récupérez des données détaillées en quelques étapes, selon les besoins
- Utilisez Amazon S3 ou DynamoDB pour les données volumineuses, transmettez les références dans l'état
- Évitez de transmettre de grosses charges utiles entre les étapes

TypeScript

```
import { withDurableExecution, DurableContext } from '@aws/durable-execution-sdk-
js';

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    // Store only the order ID, not the full order object
    const orderId = event.orderId;

    // Fetch data within each step as needed
    await context.step('validate-order', async () => {
      const order = await orderService.getOrder(orderId);
      return validateOrder(order);
    });

    await context.step('process-payment', async () => {
      const order = await orderService.getOrder(orderId);
      return processPayment(order);
    });

    return { statusCode: 200, orderId };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import durable_execution, DurableContext

@durable_execution
def handler(event, context: DurableContext):
    # Store only the order ID, not the full order object
    order_id = event['orderId']

    # Fetch data within each step as needed
    context.step(
        lambda _: validate_order(order_service.get_order(order_id)),
        name='validate-order'
    )

    context.step(
        lambda _: process_payment(order_service.get_order(order_id)),
        name='process-payment'
```

```
    )

    return {'statusCode': 200, 'orderId': order_id}
```

Concevez des étapes efficaces

Les étapes sont l'unité de travail fondamentale des fonctions durables. Des étapes bien conçues facilitent la compréhension, le débogage et la maintenance des flux de travail.

Principes de conception des étapes :

- Utilisez des noms descriptifs : les noms tels que `validate-order` « `step1` instead » facilitent la compréhension des journaux et des erreurs
- Conservez les noms statiques : n'utilisez pas de noms dynamiques avec des horodatages ou des valeurs aléatoires. Les noms des étapes doivent être déterministes pour la rediffusion
- Équilibrez la granularité : divisez les opérations complexes en étapes ciblées, mais évitez les petites étapes excessives qui augmentent la charge des points de contrôle
- Opérations liées au groupe - Les opérations qui devraient réussir ou échouer ensemble appartiennent à la même étape

Utilisez efficacement les opérations d'attente

Les opérations d'attente suspendent l'exécution sans consommer de ressources ni entraîner de coûts. Utilisez-les au lieu de faire fonctionner Lambda.

Attentes basées sur le temps : à utiliser `context.wait()` pour les retards au lieu de `setTimeout` ou. `sleep`

Rappels externes : à utiliser `context.waitForCallback()` lorsque vous attendez des systèmes externes. Définissez toujours des délais d'attente pour éviter les temps d'attente indéfinis.

Sondage : `context.waitForCondition()` à utiliser avec un recul exponentiel pour interroger les services externes sans les surcharger.

TypeScript

```
// Wait 24 hours without cost
await context.wait({ seconds: 86400 });

// Wait for external callback with timeout
const result = await context.waitForCallback(
  'external-job',
  async (callbackId) => {
    await externalService.submitJob({
      data: event.data,
      webhookUrl: `https://api.example.com/callbacks/${callbackId}`
    });
  },
  { timeout: { seconds: 3600 } }
);
```

Python

```
# Wait 24 hours without cost
context.wait(86400)

# Wait for external callback with timeout
result = context.wait_for_callback(
    lambda callback_id: external_service.submit_job(
        data=event['data'],
        webhook_url=f'https://api.example.com/callbacks/{callback_id}'
    ),
    name='external-job',
    config=WaitForCallbackConfig(timeout_seconds=3600)
)
```

Considérations supplémentaires

Gestion des erreurs : réessayez les échecs transitoires tels que les délais d'expiration du réseau et les limites de débit. Ne réessayez pas en cas d'échec permanent, comme une saisie non valide ou une erreur d'authentification. Configurez des stratégies de nouvelles tentatives avec des taux de tentatives et d'interruption maximaux appropriés. Pour des exemples détaillés, voir Gestion des erreurs et nouvelles tentatives.

Performances : minimisez la taille des points de contrôle en stockant des références plutôt que des charges utiles complètes. Utilisez `context.parallel()` et `context.map()` pour exécuter simultanément des opérations indépendantes. Opérations liées au batch pour réduire la surcharge des points de contrôle.

Gestion des versions : invoquez des fonctions avec des numéros de version ou des alias pour attribuer les exécutions à des versions de code spécifiques. Assurez-vous que les nouvelles versions du code peuvent gérer l'état des anciennes versions. Ne renommez pas les étapes et ne modifiez pas leur comportement de manière à interrompre la rediffusion.

Sérialisation : utilisez des types compatibles JSON pour les entrées et les résultats des opérations. Convertissez les dates en chaînes ISO et les objets personnalisés en objets simples avant de les transmettre à des opérations durables.

Surveillance : activez la journalisation structurée avec les noms d'exécution IDs et d'étape. Configurez CloudWatch des alarmes pour les taux d'erreur et la durée d'exécution. Utilisez le traçage pour identifier les goulots d'étranglement. Pour obtenir des instructions détaillées, consultez la section Surveillance et débogage.

Tests : testez le chemin heureux, la gestion des erreurs et le comportement des rediffusions. Testez des scénarios de temporisation pour les rappels et les temps d'attente. Utilisez des tests locaux pour réduire le temps d'itération. Pour obtenir des instructions détaillées, voir Tester des fonctions durables.

Erreurs courantes à éviter : n'imbriquez pas les `context.step()` appels, utilisez plutôt des contextes enfantins. Enveloppez les opérations non déterministes par étapes. Définissez toujours des délais d'expiration pour les rappels. Équilibrez la granularité des étapes avec la surcharge des points de contrôle. Stockez les références plutôt que les objets volumineux dans l'état.

Ressources supplémentaires

- Documentation du SDK Python - Référence complète des API, modèles de test et exemples avancés
- TypeScript Documentation du SDK : référence complète des API, modèles de test et exemples avancés

Instances gérées Lambda

Les instances gérées Lambda vous permettent d'exécuter des fonctions Lambda sur vos instances Amazon de génération actuelle, notamment Graviton4, les EC2 instances optimisées pour le réseau et d'autres options de calcul spécialisées, sans gérer les cycles de vie des instances, les correctifs du système d'exploitation et du langage d'exécution, le routage, l'équilibrage de charge ou les politiques de dimensionnement. Avec les instances gérées Lambda, vous bénéficiez d'avantages EC2 tarifaires, notamment des EC2 Savings Plans et des instances réservées.

Pour obtenir la liste des types d'instances pris en charge, rendez-vous sur la page de AWS Lambda tarification et sélectionnez votre Région AWS.

Capacités clés

Les instances gérées Lambda fournissent les fonctionnalités suivantes :

- Choisissez les instances appropriées : sélectionnez les instances appropriées en fonction des exigences en matière de performances et de coûts, notamment l'accès aux dernières technologies CPUs telles que Graviton4, les ratios mémoire-processeur configurables et les réseaux à large bande passante.
- Provisionnement automatique : AWS provisionne automatiquement les instances appropriées et active les environnements d'exécution des fonctions.
- Dimensionnement dynamique : les instances évoluent de manière dynamique en fonction des modèles de trafic de votre fonction.
- Expérience entièrement gérée : AWS gère la gestion de l'infrastructure, le dimensionnement, l'application de correctifs et le routage, avec les mêmes intégrations étendues de sources d'événements que vous connaissez bien.

Quand utiliser les instances gérées Lambda

Envisagez les instances gérées Lambda pour les cas d'utilisation suivants :

- Charges de travail élevées prévisibles en termes de volumes : idéal pour les charges de travail stables sans pics de trafic inattendus. Les instances gérées Lambda sont évolutives pour gérer le doublement du trafic en cinq minutes par défaut.

- Applications critiques en termes de performances : accès aux derniers CPUs ratios mémoire/processeur variables et à un débit réseau élevé
- Exigences réglementaires - Besoins de gouvernance granulaire avec contrôle du VPC et du placement des instances
- Variété d'applications : applications basées sur les événements, media/data traitements, applications Web et charges de travail héritées qui migrent vers le mode sans serveur

Comment ça marche

Les instances gérées Lambda utilisent des fournisseurs de capacité comme base pour exécuter vos fonctions :

1. Création d'un fournisseur de capacité : définissez où s'exécutent vos fonctions en spécifiant la configuration du VPC et, éventuellement, les exigences en matière d'instance et la configuration de dimensionnement
2. Créez votre fonction - Créez des fonctions Lambda comme d'habitude et associez-les à un fournisseur de capacité
3. Publier une version de fonction - Les versions de fonction deviennent actives sur les instances du fournisseur de capacité une fois publiées

Lorsque vous publiez une version de fonction auprès d'un fournisseur de capacité, Lambda lance Managed Instances dans votre compte. Il lance trois instances par défaut pour la résilience AZ et démarre trois environnements d'exécution avant de marquer la version ACTIVE de votre fonction. Si vous associez une fonction à un fournisseur de capacité existant qui exécute déjà d'autres fonctions, Lambda risque de ne pas créer de nouvelles instances si les instances disponibles ont déjà la capacité de s'adapter aux environnements d'exécution de la nouvelle fonction.

Modèle de simultanéité

Les instances gérées Lambda prennent en charge les invocations multiconcurrentes, dans le cadre desquelles un environnement d'exécution peut gérer plusieurs invocations en même temps. Cela diffère du type de calcul Lambda (par défaut), qui fournit un modèle de simultanéité unique dans lequel un environnement d'exécution peut exécuter au maximum un appel à la fois. La multisimultanéité permet une meilleure utilisation de vos EC2 instances sous-jacentes et est particulièrement avantageuse pour les applications gourmandes en E/S, telles que les services

Web ou les tâches par lots. Cette modification du modèle d'exécution signifie que la sécurité des threads, la gestion des états et l'isolation du contexte doivent être gérées différemment en fonction de l'environnement d'exécution.

Location et isolement

Le type de calcul Lambda (par défaut) est mutualisé, utilisant la technologie Firecracker MicroVM pour isoler les environnements d'exécution exécutés sur des flottes Lambda partagées. Les instances gérées Lambda s'exécutent sur votre compte et fournissent les dernières options EC2 matérielles et tarifaires. Les instances gérées utilisent des conteneurs exécutés sur des instances EC2 Nitro pour assurer l'isolation plutôt que sur Firecracker. Les fournisseurs de capacité servent de limite de sécurité pour les fonctions Lambda. Les fonctions s'exécutent dans des conteneurs au sein d'instances.

Comprendre les instances gérées

Les fonctions Lambda Managed Instances s'exécutent sur les instances EC2 gérées de votre compte. Ces instances sont entièrement gérées par Lambda, ce qui signifie que vous disposez d'autorisations limitées sur celles-ci par rapport aux instances standard EC2 . Vous pouvez identifier les instances gérées Lambda dans votre compte en :

- La présence du `Operator` champ en EC2 `DescribeInstances` sortie
- Le `aws:lambda:capacity-provider` tag de l'instance

Vous ne pouvez pas effectuer EC2 d'opérations standard directement sur ces instances, telles que leur arrêt manuel. Pour détruire les instances gérées, supprimez le fournisseur de capacité associé. Lambda mettra ensuite fin aux instances dans le cadre du processus de suppression du fournisseur de capacité.

Tarification

Les instances gérées Lambda utilisent une tarification EC2 basée sur des frais de gestion de 15 % en plus du coût de l' EC2 instance. Ce modèle de tarification prend en charge les EC2 Savings Plans, les instances réservées et toute autre remise tarifaire appliquée à votre EC2 utilisation. Reportez-vous à la page de tarification pour plus de détails : https://aws.amazon.com/lambda/prix/

Important : EC2 les remises tarifaires ne s'appliquent qu'au EC2 calcul sous-jacent, et non aux frais de gestion.

En quoi les instances gérées Lambda diffèrent-elles du type de calcul Lambda (par défaut)

Les instances gérées Lambda modifient la façon dont Lambda traite les demandes par rapport à Lambda (par défaut).

Principales différences :

	Lambda (par défaut)	Instances gérées Lambda
Modèle de simultanéité	Modèle de simultanéité unique dans lequel un environnement d'exécution peut prendre en charge un maximum d'un appel à la fois	Appels multisimultanés dans lesquels un environnement d'exécution peut gérer plusieurs appels simultanément, ce qui augmente le débit, en particulier pour les applications gourmandes en E/S
Location et isolement	Multitenant, utilisant la technologie Firecracker MicroVM pour isoler les environnements d'exécution exécutés sur des flottes Lambda partagées	Exécutez sur votre compte en utilisant EC2 Nitro pour vous isoler. Les fournisseurs de capacité servent de limite de sécurité, les fonctions s'exécutant dans des conteneurs au sein des instances
Modèle de tarification	Tarification par durée de traitement par demande	Tarification basée sur les instances avec des modèles de EC2 tarification, y compris les instances réservées et à la demande, et des options d'économie telles que Compute Savings Plans
Comportement de mise à l’échelle.	S'adapte lorsqu'il n'existe aucun environnement d'exécution libre pour gérer un appel entrant	S'adapte de manière asynchrone en fonction de l'utilisation des ressources du processeur

	Lambda (par défaut)	Instances gérées Lambda
	(démarrage à froid). Échelle jusqu'à zéro sans trafic	uniquement, sans démarrage à froid. S'adapte à des environnements d'exécution minimale configurés sans trafic
Le mieux adapté pour	Fonctionnalités soumises à un trafic intense qui peuvent supporter un certain temps de démarrage à froid, ou applications sans charge soutenue qui bénéficient d'une mise à l'échelle nulle	Le trafic à volume élevé et prévisible fonctionne lorsque vous recherchez la flexibilité, les plans tarifaires et les options matérielles de EC2

Étapes suivantes

- En savoir plus sur les fournisseurs de capacité pour les instances gérées Lambda
- Comprendre le dimensionnement pour les instances gérées Lambda
- Consultez les guides spécifiques à l'exécution pour Java, Node.js et Python
- Configurez la connectivité VPC pour vos fournisseurs de capacité
- Comprendre la sécurité et les autorisations pour les instances gérées Lambda

Commencer à utiliser les instances gérées Lambda

Création d'une fonction d'instance gérée Lambda (console)

Vous pouvez utiliser la console Lambda pour créer une fonction d'instance gérée qui s'exécute sur des EC2 instances Amazon gérées par un fournisseur de capacité.

Important : Avant de créer une fonction d'instance gérée, vous devez d'abord créer un fournisseur de capacité. Ces fonctions nécessitent un fournisseur de capacité pour définir l' EC2 infrastructure Amazon qui exécutera vos fonctions.

Pour créer une fonction d'instance gérée Lambda (console)

1. Ouvrez la console Lambda.
2. Choisissez Capacity providers dans le volet de navigation de gauche.
3. Choisissez Créer un fournisseur de capacité.
4. Dans la section Paramètres du fournisseur de capacité, entrez le nom de votre fournisseur de capacité.
5. Sélectionnez le VPC et les autorisations pour votre fournisseur de capacité. Vous pouvez utiliser un existant ou en créer un nouveau. Pour plus d'informations sur la création du rôle d'opérateur requis, voir Rôle d'opérateur Lambda pour les instances gérées Lambda.
6. Développez Advanced settings (Paramètres avancés).
7. Définissez les exigences de votre instance en choisissant l'architecture du processeur et les types d'instance.
8. Sous Mise à l'échelle automatique, spécifiez le nombre maximal de EC2 v CPUs pour votre fournisseur de capacité. Vous pouvez également choisir le mode de dimensionnement manuel de l'instance pour définir votre propre valeur de mise à l'échelle pour un contrôle précis.
9. Choisissez Créer un fournisseur de capacité pour en créer un nouveau.
10. Choisissez ensuite Create function.
11. Sélectionnez Créer à partir de zéro.
12. Dans le volet Informations de base, indiquez le nom de la fonction.
13. Pour Runtime, choisissez l'un des Runtimes pris en charge.
14. Choisissez l'architecture de votre fonction (identique à celle que vous avez sélectionnée pour le fournisseur de capacité). Par défaut, x86_64.
15. Sous Autorisations, assurez-vous d'avoir l'autorisation pour le rôle d'exécution choisi. Sinon, vous pouvez créer un nouveau rôle.
16. Sous Configurations supplémentaires, sélectionnez le type de calcul en tant qu'instances gérées Lambda.
17. L'ARN du fournisseur de capacité que vous avez créé au cours des étapes précédentes doit être présélectionné.
18. Choisissez la taille de la mémoire et le ratio de mémoire de l'environnement d'exécution (GiB) par vCPU.
19. Choisissez Créer une fonction.

Votre fonction d'instance gérée Lambda est créée et fournira de la capacité au fournisseur de capacité que vous avez spécifié. La création d'une fonction prend généralement plusieurs minutes. Une fois terminé, vous pouvez modifier le code de votre fonction et exécuter votre premier test.

Création d'une fonction d'instance gérée Lambda (CLI)AWS

Conditions préalables

Avant de commencer, assurez-vous de disposer des éléments suivants :

- AWS CLI — Installez et configurez la AWS CLI. Pour plus d'informations, voir Installation ou mise à jour de la dernière version de la AWS CLI.
- Autorisations IAM : votre utilisateur ou rôle IAM doit être autorisé à créer des fonctions Lambda, à fournir des capacités et à transmettre des rôles IAM. Notez que vous en aurez également besoin `iam:CreateServiceLinkedRole` si c'est la première fois que vous créez un fournisseur de capacité dans le compte ou si le rôle lié au service (SLR) a été supprimé.

Étape 1 : créer les rôles IAM requis

Les instances gérées Lambda nécessitent deux rôles IAM : un rôle d'exécution pour votre fonction et un rôle d'opérateur pour le fournisseur de capacité. Le rôle d'opérateur permet à Lambda de lancer, de résilier et de surveiller les EC2 instances Amazon en votre nom. Le rôle d'exécution de la fonction accorde à la fonction l'autorisation d'accéder à d'autres AWS services et ressources.

Pour créer le rôle d'exécution Lambda

1. Créez un document de politique de confiance qui permet à Lambda d'assumer le rôle :

```
cat > lambda-trust-policy.json << 'EOF'
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
```

```
}
EOF
```

2. Créez le rôle d'exécution :

```
aws iam create-role \
  --role-name MyLambdaExecutionRole \
  --assume-role-policy-document file://lambda-trust-policy.json
```

3. Joignez la politique d'exécution de base :

```
aws iam attach-role-policy \
  --role-name MyLambdaExecutionRole \
  --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
```

Pour créer le rôle d'opérateur fournisseur de capacité

1. Créez un document de politique de confiance qui permet à Lambda d'assumer le rôle d'opérateur :

```
cat > operator-trust-policy.json << 'EOF'
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
EOF
```

2. Créez le rôle d'opérateur :

```
aws iam create-role \
  --role-name MyCapacityProviderOperatorRole \
  --assume-role-policy-document file://operator-trust-policy.json
```

3. Joignez la politique EC2 d'autorisation requise :

```
aws iam attach-role-policy \
  --role-name MyCapacityProviderOperatorRole \
  --policy-arn arn:aws:iam::aws:policy/AWSLambdaManagedEC2ResourceOperator
```

Étape 2 : configurer les ressources VPC

Les instances gérées Lambda s'exécutent dans votre VPC et nécessitent un sous-réseau et un groupe de sécurité.

Pour créer des ressources VPC

1. Créez un VPC :

```
VPC_ID=$(aws ec2 create-vpc \
  --cidr-block 10.0.0.0/16 \
  --query 'Vpc.VpcId' \
  --output text)
```

2. Créez un sous-réseau :

```
SUBNET_ID=$(aws ec2 create-subnet \
  --vpc-id $VPC_ID \
  --cidr-block 10.0.1.0/24 \
  --query 'Subnet.SubnetId' \
  --output text)
```

3. Créez un groupe de sécurité :

```
SECURITY_GROUP_ID=$(aws ec2 create-security-group \
  --group-name my-capacity-provider-sg \
  --description "Security group for Lambda Managed Instances" \
  --vpc-id $VPC_ID \
  --query 'GroupId' \
  --output text)
```

Remarque : Les fonctions de vos instances gérées Lambda nécessitent une configuration VPC pour accéder à des ressources extérieures au VPC et pour transmettre des données de télémétrie à Logs and X-Ray. CloudWatch Pour plus de détails sur la configuration, consultez la section Mise en réseau pour les instances gérées Lambda.

Étape 3 : Créer un fournisseur de capacité

Un fournisseur de capacité gère les EC2 instances qui exécutent vos fonctions Lambda.

Pour créer un fournisseur de capacité

```
ACCOUNT_ID=$(aws sts get-caller-identity --query Account --output text)

aws lambda create-capacity-provider \
  --capacity-provider-name my-capacity-provider \
  --vpc-config SubnetIds=[$SUBNET_ID],SecurityGroupIds=[$SECURITY_GROUP_ID] \
  --permissions-config CapacityProviderOperatorRoleArn=arn:aws:iam::${ACCOUNT_ID}:role/
MyCapacityProviderOperatorRole \
  --instance-requirements Architectures=[x86_64] \
  --capacity-provider-scaling-config MaxVCpuCount=30
```

Cette commande crée un fournisseur de capacité avec la configuration suivante :

- Configuration VPC — Spécifie le sous-réseau et le groupe de sécurité pour les instances EC2
- Autorisations — Définit le rôle IAM que Lambda utilise pour gérer les instances EC2
- Exigences relatives à l'instance — Spécifie l'architecture x86_64
- Configuration de mise à l'échelle : définit un maximum de 30 V CPUs pour le fournisseur de capacité

Étape 4 : Création d'une fonction Lambda avec du code en ligne

Pour créer une fonction avec du code en ligne

1. Tout d'abord, créez une fonction Python simple et empaquez-la en ligne :

```
# Create a temporary directory for the function code
mkdir -p /tmp/my-lambda-function
cd /tmp/my-lambda-function

# Create a simple Python handler
cat > lambda_function.py << 'EOF'
import json

def lambda_handler(event, context):
    return {
```

```
        'statusCode': 200,
        'body': json.dumps({
            'message': 'Hello from Lambda Managed Instances!',
            'event': event
        })
    }
EOF

# Create a ZIP file
zip function.zip lambda_function.py
```

2. Créez la fonction Lambda à l'aide du fichier ZIP intégré :

```
ACCOUNT_ID=$(aws sts get-caller-identity --query Account --output text)
REGION=$(aws configure get region)

aws lambda create-function \
  --function-name my-managed-instance-function \
  --package-type Zip \
  --runtime python3.13 \
  --handler lambda_function.lambda_handler \
  --zip-file fileb:///tmp/my-lambda-function/function.zip \
  --role arn:aws:iam::${ACCOUNT_ID}:role/MyLambdaExecutionRole \
  --architectures x86_64 \
  --memory-size 2048 \
  --ephemeral-storage Size=512 \
  --capacity-provider-config
 LambdaManagedInstancesCapacityProviderConfig={CapacityProviderArn=arn:aws:lambda:
${REGION}:${ACCOUNT_ID}:capacity-provider:my-capacity-provider}
```

La fonction est créée avec :

- Runtime — Python 3.13
- Handler — La `lambda_handler` fonction dans `lambda_function.py`
- Mémoire — 2048 Mo
- Stockage éphémère — 512 Mo
- Fournisseur de capacité : liens vers le fournisseur de capacité que vous avez créé

Étape 5 : Publier une version de fonction

Pour exécuter votre fonction sur des instances gérées Lambda, vous devez publier une version.

Pour publier une version de fonction

```
aws lambda publish-version \
  --function-name my-managed-instance-function
```

Cette commande publie la version 1 de votre fonction et la déploie auprès du fournisseur de capacité.

Étape 6 : Invoquez votre fonction

Après la publication, vous pouvez appeler votre fonction.

Pour appeler votre fonction

```
aws lambda invoke \
  --function-name my-managed-instance-function:1 \
  --payload '{"name": "World"}' \
  response.json

# View the response
cat response.json
```

La fonction s'exécute sur les EC2 instances gérées par votre fournisseur de capacité et renvoie une réponse.

Nettoyage

Pour éviter d'encourir des frais, supprimez les ressources que vous avez créées :

1. Supprimez la fonction :

   ```
   aws lambda delete-function --function-name my-managed-instance-function
   ```

2. Supprimez le fournisseur de capacité :

   ```
   aws lambda delete-capacity-provider --capacity-provider-name my-capacity-provider
   ```

3. Supprimez les ressources VPC :

   ```
   aws ec2 delete-security-group --group-id $SECURITY_GROUP_ID
   aws ec2 delete-subnet --subnet-id $SUBNET_ID
   ```

```
aws ec2 delete-vpc --vpc-id $VPC_ID
```

4. Supprimez les rôles IAM :

```
aws iam detach-role-policy \
  --role-name MyLambdaExecutionRole \
  --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
aws iam detach-role-policy \
  --role-name MyCapacityProviderOperatorRole \
  --policy-arn arn:aws:iam::aws:policy/AWSLambdaManagedEC2ResourceOperator

aws iam delete-role --role-name MyLambdaExecutionRole
aws iam delete-role --role-name MyCapacityProviderOperatorRole
```

Concepts de base

Les instances gérées par Lambda introduisent plusieurs concepts fondamentaux qui diffèrent des fonctions Lambda traditionnelles. Comprendre ces concepts est essentiel pour déployer et gérer efficacement vos fonctions sur EC2 l'infrastructure.

Les fournisseurs de capacité constituent la base des instances gérées Lambda. Un fournisseur de capacité définit l'infrastructure informatique dans laquelle s'exécutent vos fonctions, y compris la configuration du VPC, les exigences relatives aux instances et les politiques de dimensionnement. Les fournisseurs de capacité servent également de limite de sécurité pour vos fonctions, ce qui signifie que toutes les fonctions assignées au même fournisseur de capacité doivent être mutuellement fiables.

Le comportement de mise à l'échelle est très différent de celui des fonctions Lambda traditionnelles. Au lieu de s'adapter à la demande lorsque les appels arrivent, les instances gérées évoluent de manière asynchrone en fonction de l'utilisation des ressources du processeur. Cette approche élimine les démarrages à froid, mais nécessite une planification en fonction de la croissance du trafic. Si votre trafic double plus que dans les 5 minutes, vous risquez de rencontrer des difficultés à mesure que Lambda augmente sa capacité pour répondre à la demande.

La sécurité et les autorisations nécessitent une attention particulière. Vous avez besoin d'autorisations de rôle d'opérateur pour permettre à Lambda de gérer les EC2 ressources de vos fournisseurs de capacité. En outre, les utilisateurs doivent être `lambda:PassCapacityProvider` autorisés à attribuer des fonctions aux fournisseurs de capacité, agissant comme une barrière de sécurité pour contrôler quelles fonctions peuvent être exécutées sur une infrastructure spécifique.

L'exécution multisimultanée est une caractéristique clé des instances gérées. Chaque environnement d'exécution peut gérer plusieurs appels simultanément, maximisant ainsi l'utilisation des ressources pour les applications gourmandes en E/S. Cela diffère du Lambda traditionnel où chaque environnement traite une demande à la fois. Ce modèle d'exécution nécessite une attention particulière à la sécurité des threads, à la gestion des états et à l'isolation du contexte en fonction de votre environnement d'exécution.

Les sections suivantes fournissent des informations détaillées sur chaque concept de base.

Fournisseurs de capacité

Un fournisseur de capacité est la base de l'exécution des instances gérées Lambda. Il sert de limite de sécurité pour vos fonctions et définit les ressources de calcul que Lambda fournira et gérera en votre nom.

Lorsque vous créez un fournisseur de capacité, vous spécifiez :

- Configuration VPC : sous-réseaux et groupes de sécurité sur lesquels les instances seront exécutées
- Autorisations : rôles IAM permettant à Lambda de gérer les ressources EC2
- Exigences relatives à l'instance (facultatif) : préférences en matière d'architecture et de type d'instance
- Configuration de dimensionnement (facultatif) - Comment Lambda fait évoluer vos instances

Comprendre les fournisseurs de capacité comme une limite de sécurité

Les fournisseurs de capacité servent de limite de sécurité pour les fonctions Lambda au sein de votre VPC, remplaçant ainsi l'isolation basée sur FireCracker. Les fonctions s'exécutent dans des conteneurs au sein des instances, mais les conteneurs ne fournissent pas une isolation de sécurité solide entre les fonctions, contrairement à Firecracker micro. VMs

Concepts de sécurité clés :

- Fournisseur de capacité : limite de sécurité qui définit les niveaux de confiance pour les fonctions Lambda
- Isolation des conteneurs : les conteneurs ne sont PAS un fournisseur de sécurité. Ne comptez pas sur eux pour assurer la sécurité entre des charges de travail non fiables

- Séparation de confiance : Séparez les charges de travail qui ne sont pas mutuellement fiables en utilisant différents fournisseurs de capacité

Création d'un fournisseur de capacité

Vous pouvez créer un fournisseur de capacité à l'aide de la AWS CLI, AWS de la console de gestion ou AWS SDKs.

À l'aide de la AWS CLI :

```
aws lambda create-capacity-provider \
  --capacity-provider-name my-capacity-provider \
  --vpc-config
 SubnetIds=subnet-12345,subnet-67890,subnet-11111,SecurityGroupIds=sg-12345 \
  --permissions-config CapacityProviderOperatorRoleArn=arn:aws:iam::123456789012:role/
MyOperatorRole \
  --instance-requirements Architectures=x86_64 \
  --capacity-provider-scaling-config ScalingMode=Auto
```

Paramètres requis

CapacityProviderName

- Un nom unique pour votre fournisseur de capacité
- Doit être unique au sein de votre AWS compte

VpcConfig

- SubnetIds(obligatoire) : Au moins un sous-réseau, 16 au maximum. Utilisez des sous-réseaux répartis sur plusieurs zones de disponibilité pour renforcer la résilience
- SecurityGroupIds(facultatif) : groupes de sécurité pour vos instances. Par défaut, c'est le groupe de sécurité par défaut du VPC s'il n'est pas spécifié

PermissionsConfig

- CapacityProviderOperatorRoleArn(obligatoire) : rôle IAM qui permet à Lambda de EC2 gérer les ressources de votre fournisseur de capacités

Paramètres facultatifs

InstanceRequirements

Spécifiez l'architecture et les types d'instances de votre fournisseur de capacité :

- Architectures : Choisissez `x86_64` ou`arm64`. La valeur par défaut est `x86_64`
- AllowedInstanceTypes: Spécifiez les types d'instances autorisés. Exemple : `m5.8xlarge`
- ExcludedInstanceTypes: Spécifiez les types d'instances exclus à l'aide de caractères génériques. Vous ne pouvez spécifier qu'un AllowedInstanceTypes ou ExcludedInstanceTypes

Par défaut, Lambda choisit les types d'instances les mieux adaptés à votre charge de travail. Nous vous recommandons de laisser les instances gérées Lambda choisir les types d'instances à votre place, car la restriction du nombre de types d'instances possibles peut entraîner une baisse de disponibilité.

CapacityProviderScalingConfig

Configurez la manière dont Lambda fait évoluer vos instances :

- ScalingMode: Réglez sur `Auto` pour une mise à l'échelle automatique ou `Manual` pour un contrôle manuel. La valeur par défaut est `Auto`
- VCpuNombre maximum : nombre maximal de v CPUs pour le fournisseur de capacité. La valeur par défaut est 400.
- ScalingPolicies: Définissez des politiques de dimensionnement du suivi des cibles pour l'utilisation du processeur et de la mémoire

KmsKeyArn

Spécifiez une AWS KMS clé pour le chiffrement EBS. La valeur par défaut est la clé AWS gérée si elle n'est pas spécifiée.

Balises

Ajoutez des balises pour organiser et gérer vos fournisseurs de capacités.

Gestion des fournisseurs de capacités

Mise à jour d’un fournisseur de capacité

Vous pouvez mettre à jour certaines propriétés d'un fournisseur de capacité à l'aide de l'`UpdateCapacityProvider`API.

```
aws lambda update-capacity-provider \
  --capacity-provider-name my-capacity-provider \
  --capacity-provider-scaling-config ScalingMode=Auto
```

Suppression d’un fournisseur de capacité

Vous pouvez supprimer un fournisseur de capacité lorsqu'il n'est plus nécessaire à l'aide de l'`DeleteCapacityProvider`API.

```
aws lambda delete-capacity-provider \
  --capacity-provider-name my-capacity-provider
```

Remarque : vous ne pouvez pas supprimer un fournisseur de capacité auquel des versions de fonctions sont associées.

Afficher les détails du fournisseur de capacité

Récupérez des informations sur un fournisseur de capacité à l'aide de l'`GetCapacityProvider`API.

```
aws lambda get-capacity-provider \
  --capacity-provider-name my-capacity-provider
```

États fournisseurs de capacités

Un fournisseur de capacité peut se trouver dans l'un des états suivants :

- En attente : le fournisseur de capacité est en cours de création
- Actif : le fournisseur de capacité est prêt à être utilisé
- Échec : échec de la création du fournisseur de capacité
- Suppression : le fournisseur de capacité est en cours de suppression

Quotas

- Capacité maximale de fournisseurs par compte : 1 000
- Nombre maximum de versions de fonctions par fournisseur de capacité : 100 (ne peut pas être augmenté)

Bonnes pratiques

1. Séparation par niveau de confiance : créez différents fournisseurs de capacité pour les charges de travail ayant des exigences de sécurité différentes
2. Utilisez des noms descriptifs : nommez les fournisseurs de capacité pour indiquer clairement leur utilisation prévue et leur niveau de confiance (par exemple`production-trusted`,`dev-sandbox`)
3. Utiliser plusieurs zones de disponibilité : Spécifiez des sous-réseaux sur plusieurs AZs pour une haute disponibilité
4. Laissez Lambda choisir les types d'instances : sauf si vous avez des exigences matérielles spécifiques, permettez à Lambda de sélectionner les meilleurs types d'instances pour une disponibilité optimale
5. Surveiller l'utilisation : AWS CloudTrail à utiliser pour surveiller les attributions des fournisseurs de capacité et les modèles d'accès

Étapes suivantes

- En savoir plus sur le dimensionnement des instances gérées Lambda
- Comprendre la sécurité et les autorisations pour les instances gérées Lambda
- Configurez la connectivité VPC pour vos fournisseurs de capacité
- Consultez les guides spécifiques à l'exécution pour Java, Node.js et Python

Dimensionnement des instances gérées Lambda

Les instances gérées Lambda ne sont pas évolutives lorsque les appels arrivent et ne prennent pas en charge les démarrages à froid. Il évolue plutôt de manière asynchrone en utilisant des signaux de consommation de ressources. Les instances gérées évoluent actuellement en fonction de l'utilisation des ressources du processeur et de la saturation en multisimultanéité.

Principales différences :

- Lambda (par défaut) : évolue lorsqu'il n'existe aucun environnement d'exécution libre pour gérer un appel entrant (démarrage à froid)
- Instances gérées Lambda : évolue de manière asynchrone en fonction de l'utilisation des ressources du processeur et de la saturation multiconcurrentielle des environnements d'exécution

Si votre trafic double plus que dans les 5 minutes, vous risquez de rencontrer des difficultés à mesure que Lambda adapte les instances et les environnements d'exécution pour répondre à la demande.

Le cycle de vie du dimensionnement

Les instances gérées Lambda utilisent une architecture distribuée pour gérer le dimensionnement :

Composants :

- Instances gérées - Exécutez-les dans votre compte dans les sous-réseaux que vous fournissez
- Routeur et scaleur : composants Lambda partagés qui acheminent les appels et gèrent le dimensionnement
- Agent Lambda : s'exécute sur chaque instance gérée pour gérer le cycle de vie de l'environnement d'exécution et surveiller la consommation de ressources

Comment cela fonctionne :

1. Lorsque vous publiez une version de fonction auprès d'un fournisseur de capacité, Lambda lance Managed Instances dans votre compte. Il en lance trois par défaut pour la résilience AZ et démarre trois environnements d'exécution avant de marquer la version ACTIVE de votre fonction.
2. Chaque instance gérée peut exécuter des environnements d'exécution pour plusieurs fonctions mappées vers le même fournisseur de capacité.
3. À mesure que le trafic entre dans votre application, les environnements d'exécution consomment des ressources. L'agent Lambda informe le scaler, qui décide s'il convient de dimensionner de nouveaux environnements d'exécution ou des instances gérées.
4. Si Router tente d'envoyer un appel à un environnement d'exécution consommant beaucoup de ressources, l'agent Lambda de cette instance l'invite à réessayer sur une autre instance.
5. Lorsque le trafic diminue, l'agent Lambda en informe Scaler, qui prend la décision de réduire les environnements d'exécution et de dimensionner les instances gérées.

Ajustement du comportement de dimensionnement

Vous pouvez personnaliser le comportement de dimensionnement des instances gérées à l'aide de quatre commandes :

Contrôles du niveau de fonction

1. Fonction : mémoire et v CPUs

Choisissez la taille de mémoire et l'allocation de vCPU pour votre fonction. La plus petite taille de fonction prise en charge est de 2 Go et 1 vCPU.

Considérations :

- Choisissez un paramètre de mémoire et de vCPU qui prendra en charge les exécutions simultanées de votre fonction
- Vous ne pouvez pas configurer une fonction avec moins d'un vCPU car les fonctions exécutées sur des instances gérées doivent prendre en charge des charges de travail simultanées multiples
- Vous ne pouvez pas choisir moins de 2 Go car cela correspond au ratio mémoire/vCPU de 2 pour 1 des instances C, qui ont le ratio le plus faible
- Pour les applications Python, vous devrez peut-être choisir un ratio mémoire/v plus élevéCPUs, par exemple 4 pour 1 ou 8 pour 1, en raison de la façon dont Python gère la multisimultanéité
- Si vous exécutez des opérations gourmandes en ressources CPU ou si vous effectuez peu d'E/S, vous devez choisir plusieurs vCPU

2. Simultanéité maximum

Définissez la simultanéité maximale par environnement d'exécution.

Comportement par défaut : Lambda choisit des valeurs par défaut judicieuses qui équilibrent la consommation de ressources et le débit et conviennent à une grande variété d'applications.

Directives de réglage :

- Augmenter la simultanéité : si vos appels de fonctions utilisent très peu de CPU, vous pouvez augmenter la simultanéité maximale jusqu'à 64 par vCPU
- Diminuez la simultanéité : si votre application consomme une grande quantité de mémoire et très peu de processeur, vous pouvez réduire votre simultanéité maximale

Important : étant donné que les instances gérées Lambda sont destinées à des applications multiconcurrentes, les environnements d'exécution présentant une très faible simultanéité peuvent être limités lors du dimensionnement.

Contrôles au niveau du fournisseur de capacité

3. Utilisation des ressources cibles

Choisissez votre propre objectif en matière de consommation d'utilisation du processeur.

Comportement par défaut : Lambda conserve une marge de manœuvre suffisante pour que votre trafic double en 5 minutes sans limitation.

Options d'optimisation :

- Si votre charge de travail est très stable ou si votre application n'est pas sensible aux limitations, vous pouvez définir l'objectif à un niveau élevé pour augmenter le taux d'utilisation et réduire les coûts
- Si vous souhaitez conserver une marge de manœuvre pour faire face à des pics de trafic, vous pouvez définir des objectifs de ressources à un niveau faible, ce qui nécessitera une capacité accrue

4. Sélection du type d'instance

Définissez les types d'instances autorisés ou exclus.

Comportement par défaut : Lambda choisit les types d'instances les mieux adaptés à votre charge de travail. Nous vous recommandons de laisser les instances gérées Lambda choisir les types d'instances à votre place, car la restriction du nombre de types d'instances possibles peut entraîner une baisse de disponibilité.

Configuration personnalisée :

- Exigences matérielles spécifiques : définissez les types d'instances autorisés dans une liste d'instances compatibles. Par exemple, si votre application nécessite une bande passante réseau élevée, vous pouvez sélectionner plusieurs types d'instances n
- Optimisation des coûts : pour les environnements de test ou de développement, vous pouvez choisir des types d'instances plus petits, tels que les types d'instance m7a.large

Étapes suivantes

- En savoir plus sur les fournisseurs de capacité pour les instances gérées Lambda
- Consultez les guides spécifiques à l'exécution pour gérer la multisimultanéité
- Configurez la connectivité VPC pour vos fournisseurs de capacité
- Surveillez les métriques de dimensionnement pour optimiser le comportement de dimensionnement

Sécurité et autorisations

Les instances gérées Lambda utilisent des fournisseurs de capacité comme limites de confiance. Les fonctions s'exécutent dans des conteneurs au sein de ces instances, mais les conteneurs ne fournissent pas d'isolation de sécurité entre les charges de travail. Toutes les fonctions assignées au même fournisseur de capacité doivent être mutuellement fiables.

Concepts de sécurité clés

- Fournisseur de capacité : limite de sécurité qui définit les niveaux de confiance pour les fonctions Lambda
- Isolation des conteneurs : les conteneurs ne constituent pas une limite de sécurité. Ne vous fiez pas à eux pour assurer la sécurité entre des charges de travail non fiables
- Séparation de confiance : Séparez les charges de travail qui ne sont pas mutuellement fiables en utilisant différents fournisseurs de capacité

Autorisations nécessaires

PassCapacityProvider Action

Les utilisateurs doivent être `lambda:PassCapacityProvider` autorisés à attribuer des fonctions aux fournisseurs de capacité. Cette autorisation agit comme une barrière de sécurité, garantissant que seuls les utilisateurs autorisés peuvent confier des fonctions à des fournisseurs de capacité spécifiques.

Les administrateurs de compte contrôlent les fonctions qui peuvent utiliser des fournisseurs de capacité spécifiques par le biais de l'action `lambda:PassCapacityProvider` IAM. Cette action est requise lorsque :

- Création de fonctions utilisant des instances gérées Lambda

- Mettre à jour les configurations des fonctions pour utiliser un fournisseur de capacité
- Déploiement de fonctions via une infrastructure sous forme de code

Exemple de politique IAM

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "lambda:PassCapacityProvider",
      "Resource": "arn:aws:lambda:*:*:capacity-provider:trusted-workloads-*"
    }
  ]
}
```

Rôle lié à un service

AWS Lambda utilise le rôle `AWSServiceRoleForLambda` lié au service pour gérer les ressources ec2 des instances gérées Lambda auprès de vos fournisseurs de capacité.

Bonnes pratiques

1. Séparation par niveau de confiance : créez différents fournisseurs de capacité pour les charges de travail ayant des exigences de sécurité différentes
2. Utiliser des noms descriptifs : nommez les fournisseurs de capacité pour indiquer clairement leur utilisation prévue et leur niveau de confiance (par exemple`production-trusted`,`dev-sandbox`)
3. Appliquer le moindre privilège : octroyer `PassCapacityProvider` des autorisations uniquement aux fournisseurs de capacité nécessaires
4. Surveiller l'utilisation : AWS CloudTrail à utiliser pour surveiller les attributions des fournisseurs de capacité et les modèles d'accès

Étapes suivantes

- En savoir plus sur les fournisseurs de capacité pour les instances gérées Lambda
- Comprendre le dimensionnement pour les instances gérées Lambda
- Configurez la connectivité VPC pour vos fournisseurs de capacité

- Consultez les guides spécifiques à l'exécution pour Java, Node.js et Python

Rôle d'opérateur Lambda pour les instances gérées Lambda

Lorsque vous utilisez des instances gérées Lambda, Lambda a besoin d'autorisations pour gérer la capacité de calcul de votre compte. Le rôle d'opérateur fournit ces autorisations via des politiques IAM qui permettent à Lambda de EC2 gérer les instances du fournisseur de capacité.

Lambda assume le rôle d'opérateur lors de l'exécution de ces opérations de gestion, de la même manière que Lambda assume un rôle d'exécution lorsque votre fonction s'exécute.

Création d'un rôle d'opérateur

Vous pouvez créer un rôle d'opérateur dans la console IAM ou à l'aide de la AWS CLI. Le rôle doit inclure :

- Politique d'autorisations : accorde des autorisations pour gérer les fournisseurs de capacité et les ressources associées
- Politique de confiance — Permet au service Lambda (`lambda.amazonaws.com`) d'assumer le rôle

Politique d’autorisations

Le rôle d'opérateur nécessite des autorisations pour gérer les fournisseurs de capacité et les ressources informatiques sous-jacentes. Au minimum, le rôle nécessite les autorisations définies dans la politique AWSLambdaEC2ResourceOperatorgérée, actuellement :

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ec2:RunInstances",
        "ec2:CreateTags",
        "ec2:AttachNetworkInterface"
      ],
      "Resource": [
        "arn:aws:ec2:*:*:instance/*",
        "arn:aws:ec2:*:*:network-interface/*",
```

```
        "arn:aws:ec2:*:*:volume/*"
      ],
      "Condition": {
        "StringEquals": {
          "ec2:ManagedResourceOperator": "scaler.lambda.amazonaws.com"
        }
      }
    },
    {
      "Effect": "Allow",
      "Action": [
        "ec2:DescribeAvailabilityZones",
        "ec2:DescribeCapacityReservations",
        "ec2:DescribeInstances",
        "ec2:DescribeInstanceStatus",
        "ec2:DescribeInstanceTypeOfferings",
        "ec2:DescribeInstanceTypes",
        "ec2:DescribeSecurityGroups",
        "ec2:DescribeSubnets"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "ec2:RunInstances",
        "ec2:CreateNetworkInterface"
      ],
      "Resource": [
        "arn:aws:ec2:*:*:subnet/*",
        "arn:aws:ec2:*:*:security-group/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "ec2:RunInstances"
      ],
      "Resource": [
        "arn:aws:ec2:*:*:image/*"
      ],
      "Condition": {
        "StringEquals": {
          "ec2:Owner": "amazon"
```

```
        }
      }
    }
  ]
}
```

Politique d’approbation

La politique de confiance permet à Lambda d'assumer le rôle d'opérateur :

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Rôle lié à un service pour les instances gérées Lambda

Pour gérer de manière responsable le cycle de vie des instances gérées Lambda, Lambda a besoin d'un accès permanent pour mettre fin aux instances gérées de votre compte. Lambda utilise un rôle lié à un service (SLR) AWS Identity and Access Management (IAM) pour effectuer ces opérations.

Création automatique : le rôle lié au service est automatiquement créé la première fois que vous créez un fournisseur de capacité. L'utilisateur qui crée le premier fournisseur de capacité doit avoir l'`iam:CreateServiceLinkedRole`autorisation pour le `lambda.amazonaws.com` principal.

Autorisations : le rôle lié au service accorde à Lambda les autorisations suivantes sur les instances gérées :

- `ec2:TerminateInstances`— Pour mettre fin aux instances à la fin de leur cycle de vie
- `ec2:DescribeInstances`— Pour énumérer les instances gérées

Suppression : vous ne pouvez supprimer ce rôle lié à un service qu'après avoir supprimé tous les fournisseurs de capacité des instances gérées Lambda de votre compte.

Pour plus d'informations sur les rôles liés à un service, consultez the section called "Utilisation des rôles liés à un service".

Comprendre l'environnement d'exécution des instances gérées Lambda

Les instances gérées Lambda fournissent un modèle de déploiement alternatif qui exécute votre code de fonction sur des EC2 instances Amazon appartenant au client, tandis que Lambda gère les aspects opérationnels. L'environnement d'exécution des instances gérées présente plusieurs différences importantes par rapport aux fonctions Lambda (par défaut), notamment en ce qui concerne la façon dont il gère les appels simultanés et les cycles de vie des conteneurs.

Remarque : Pour plus d'informations sur l'environnement d'exécution Lambda (par défaut), voir Comprendre le cycle de vie de l'environnement d'exécution Lambda.

Cycle de vie des environnements d'exécution

Le cycle de vie d'un environnement d'exécution des fonctions Lambda Managed Instances diffère de celui de Lambda (par défaut) de plusieurs manières principales :

Phase d'initialisation

Pendant la phase d'initialisation, Lambda exécute les étapes suivantes :

- Initialisation et enregistrement de toutes les extensions
- Démarrez le point d'entrée du runtime. L'environnement d'exécution génère le nombre configuré de travailleurs d'exécution (l'implémentation dépend du temps d'exécution)
- Exécuter le code d'initialisation de la fonction (code en dehors du gestionnaire)
- Attendez qu'au moins un agent d'exécution indique qu'il est prêt en appelant `/runtime/invocation/next`

La phase d'initialisation est considérée comme terminée lorsque les extensions sont initialisées et qu'au moins un programme de travail d'exécution a été appelé. `/runtime/invocation/next` La fonction est alors prête à traiter les appels.

> **Note**
>
> Pour les fonctions Lambda Managed Instances, l'initialisation peut prendre jusqu'à 15 minutes. Le délai d'expiration est de 130 secondes maximum ou le délai d'expiration de la fonction configurée (jusqu'à 900 secondes).

Phase d’invocation

La phase d'appel pour les fonctions Lambda Managed Instances présente plusieurs caractéristiques uniques :

Fonctionnement en continu Contrairement à Lambda (par défaut), l'environnement d'exécution reste actif en permanence, traitant les appels au fur et à mesure qu'ils arrivent sans se figer entre les appels.

Traitement parallèle. Plusieurs invocations peuvent être exécutées simultanément dans le même environnement d'exécution, chacune étant gérée par un exécutant différent.

Délais d'attente indépendants. Le délai d'expiration configuré pour la fonction s'applique à chaque appel individuel. Lorsqu'un appel expire, Lambda marque cet appel spécifique comme ayant échoué, mais n'interrompt pas les autres invocations en cours ni ne met fin à l'environnement d'exécution.

Gestion de la contre-pression. Si tous les travailleurs d'exécution sont occupés à traiter des appels, les nouvelles demandes d'appel sont rejetées jusqu'à ce qu'un travailleur soit disponible.

Gestion des erreurs et restauration

La gestion des erreurs dans les environnements d'exécution des fonctions Lambda Managed Instances est différente de celle de Lambda (par défaut) :

Invoquez les délais d'expiration. Lorsqu'un appel individuel expire, Lambda renvoie une erreur de délai d'expiration pour cet appel. Toutefois, les instances gérées Lambda n'appliquent pas le délai d'expiration : votre code continuera de s'exécuter. En tant que développeur de fonctions, vous êtes responsable de la détection et de la gestion du délai d'expiration. L'objet de contexte indique le temps restant pour l'invocation, une valeur nulle ou négative indiquant un délai d'expiration. Les autres appels simultanés dans l'environnement d'exécution continuent à être traités normalement.

Défaillances du Runtime Worker. Si un processus de travail d'exécution se bloque, l'environnement d'exécution continue de fonctionner avec les autres travailleurs sains.

L'extension se bloque. Si un processus d'extension se bloque pendant l'initialisation ou le fonctionnement, l'ensemble de l'environnement d'exécution est marqué comme défectueux et est arrêté. Lambda crée un nouvel environnement d'exécution pour le remplacer.

Aucune réinitialisation/réparation. Contrairement à Lambda (par défaut), les instances gérées ne tentent pas de réinitialiser et de réinitialiser l'environnement d'exécution après des erreurs. Au lieu de cela, les contenants insalubres sont retirés et remplacés par de nouveaux.

Version $LATEST.PUBLISHED dans les instances gérées Lambda

Les fonctions Lambda Managed Instances prennent en charge le même flux de travail de versionnement numéroté que Lambda (par défaut). Si vous préférez ne pas conserver de versions numérotées, Lambda Managed Instances introduit un nouveau type de version :. `$LATEST.PUBLISHED` Cette version vous permet de créer ou de republier une dernière version publiée selon les besoins avec un code ou une configuration mis à jour, sans gérer les versions numérotées.

Principale différence par rapport à $LATEST : lorsque vous appelez une fonction Lambda Managed Instances à l'aide d'un ARN non qualifié, Lambda invoque implicitement la version plutôt que `$LATEST.PUBLISHED` la version non publiée $LATEST.

La commande AWS CLI suivante crée ou republie la `$LATEST.PUBLISHED` version.

```
aws lambda publish-version --function-name my-function --publish-to LATEST_PUBLISHED
```

Vous devriez voir la sortie suivante :

```
{
  "FunctionName": "my-function",
  "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function:
$LATEST.PUBLISHED",
  "Version": "$LATEST.PUBLISHED",
  "Role": "arn:aws:iam::123456789012:role/lambda-role",
  "Handler": "function.handler",
  "Runtime": "nodejs24.x",
  ...
}
```

> **Note**
>
> Si vous utilisez AWS CloudFormation la console Lambda pour créer une fonction Lambda Managed Instances, Lambda crée automatiquement la version. `$LATEST.PUBLISHED`

Runtimes des instances gérées Lambda

Lambda traite les demandes différemment lorsqu'il utilise des instances gérées par Lambda. Au lieu de traiter les demandes de manière séquentielle dans chaque environnement d'exécution, les instances gérées Lambda traitent plusieurs demandes simultanément dans chaque environnement d'exécution. Ce changement de modèle d'exécution signifie que les fonctions utilisant des instances gérées par Lambda doivent tenir compte de la sécurité des threads, de la gestion des états et de l'isolation du contexte, des problèmes qui ne se posent pas dans le modèle de simultanéité unique Lambda (par défaut). En outre, la mise en œuvre de la multisimultanéité varie selon les environnements d'exécution.

Environnements d'exécution pris en charge

Les environnements d'exécution suivants prennent en charge les instances gérées Lambda :

- Java : Java 21 et versions ultérieures.
- Python : Python 3.13 et versions ultérieures.
- Node.js : Node.js 22 et versions ultérieures.
- .NET : .NET 8 et versions ultérieures.

Considérations spécifiques à l'exécution

Chaque environnement d'exécution implémente la multisimultanéité différemment. Vous devez comprendre comment la multiconcurrence est implémentée dans le langage de programmation que vous avez choisi pour appliquer les meilleures pratiques de simultanéité appropriées.

Java

Utilise un processus unique avec des threads du système d'exploitation pour la simultanéité. Plusieurs threads exécutent simultanément la méthode du gestionnaire, ce qui nécessite une gestion sécurisée de l'état et des ressources partagées.

Python

Utilise plusieurs processus Python dans lesquels chaque demande simultanée est exécutée dans le cadre d'un processus distinct. Cela protège contre la plupart des problèmes de simultanéité, bien qu'il soit nécessaire de faire attention aux ressources partagées telles que le `/tmp` répertoire.

Node.js

Utilise des threads de travail avec une exécution asynchrone. Les demandes simultanées sont réparties entre les threads de travail, et chaque thread de travail peut également traiter les demandes simultanées de manière asynchrone, ce qui nécessite une gestion sûre de l'état et des ressources partagées.

.NET

Utilise les tâches .NET avec le traitement asynchrone de plusieurs demandes simultanées. Nécessite une gestion sûre des ressources publiques et partagées.

Étapes suivantes

Pour obtenir des informations détaillées sur chaque environnement d'exécution, consultez les rubriques suivantes :

- Runtime Java pour les instances gérées Lambda
- Runtime Node.js pour les instances gérées Lambda
- Runtime Python pour les instances gérées Lambda
- Environnement d'exécution .NET pour les instances gérées Lambda

Runtime Java pour les instances gérées Lambda

Pour les environnements d'exécution Java, les instances gérées Lambda utilisent les threads du système d'exploitation à des fins de simultanéité. Lambda charge votre objet gestionnaire une fois par environnement d'exécution lors de l'initialisation, puis crée plusieurs threads. Ces threads s'exécutent en parallèle et nécessitent une gestion sûre de l'état et des ressources partagées. Chaque thread partage le même objet de gestionnaire et tous les champs statiques.

Configuration de la simultanéité

Le nombre maximum de demandes simultanées que Lambda envoie à chaque environnement d'exécution est contrôlé par le `PerExecutionEnvironmentMaxConcurrency` paramètre de

configuration de la fonction. Ce paramètre est facultatif et la valeur par défaut varie en fonction du temps d'exécution. Pour les environnements d'exécution Java, la valeur par défaut est de 32 requêtes simultanées par vCPU, ou vous pouvez configurer votre propre valeur. Cette valeur détermine également le nombre de threads utilisés par le moteur d'exécution Java. Lambda ajuste automatiquement le nombre de demandes simultanées jusqu'au maximum configuré en fonction de la capacité de chaque environnement d'exécution à absorber ces demandes.

Création de fonctions pour la multisimultanéité

Vous devez appliquer les mêmes pratiques de sécurité des threads lorsque vous utilisez des instances gérées Lambda que dans tout autre environnement multithread. Étant donné que l'objet du gestionnaire est partagé entre tous les threads de travail d'exécution, tout état mutable doit être sûr pour les threads. Cela inclut les collections, les connexions aux bases de données et tous les objets statiques modifiés lors du traitement des demandes.

AWS Les clients du SDK sont compatibles avec les threads et ne nécessitent aucune manipulation particulière.

Exemple : pools de connexions aux bases de données

Le code suivant utilise un objet de connexion à une base de données statique partagé entre les threads. Selon la bibliothèque de connexion utilisée, cela peut ne pas être sûr pour les threads.

```
public class DBQueryHandler implements RequestHandler<Object, String> {
    // Single connection shared across all threads - NOT SAFE
    private static Connection connection;

    public DBQueryHandler() {
        this.connection = DriverManager.getConnection(jdbcUrl, username, password);
    }

    @Override
    public String handleRequest(Object input, Context context) {
        PreparedStatement stmt = connection.prepareStatement(query);
        ResultSet rs = stmt.executeQuery();
        // Multiple threads using same connection causes issues
        return result.toString();
    }
}
```

Une approche thread-safe consiste à utiliser un pool de connexions. Dans l'exemple suivant, le gestionnaire de fonctions extrait une connexion du pool. La connexion n'est utilisée que dans le cadre d'une seule demande.

```
public class DBQueryHandler implements RequestHandler<Object, String> {

    private static HikariDataSource dataSource;

    public DBQueryHandler() {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl("jdbc:mysql://localhost:3306/your_database");
        dataSource = new HikariDataSource(config); // Create pool once per Lambda
 container
    }

    @Override
    public String handleRequest(Object input, Context context) {
        String query = "SELECT column_name FROM your_table LIMIT 10";
        StringBuilder result = new StringBuilder("Data:\n");

        // try-with-resources automatically calls close() on the connection,
        // which returns it to the HikariCP pool (does NOT close the physical DB
 connection)
        try (Connection connection = dataSource.getConnection();
             PreparedStatement stmt = connection.prepareStatement(query);
             ResultSet rs = stmt.executeQuery()) {

            while (rs.next()) {
                result.append(rs.getString("column_name")).append("\n");
            }

        } catch (Exception e) {
            context.getLogger().log("Error: " + e.getMessage());
            return "Error";
        }

        return result.toString();
    }
}
```

Exemple : Collections

Les collections Java standard ne sont pas sûres pour les threads :

```
public class Handler implements RequestHandler<Object, String> {
    private static List<String> items = new ArrayList<>();
    private static Map<String, Object> cache = new HashMap<>();

    @Override
    public String handleRequest(Object input, Context context) {
        items.add("list item");  // Not thread-safe
        cache.put("key", input); // Not thread-safe
        return "Success";
    }
}
```

Utilisez plutôt des collections adaptées aux threads :

```
public class Handler implements RequestHandler<Object, String> {
    private static final List<String> items =
        Collections.synchronizedList(new ArrayList<>());
    private static final ConcurrentHashMap<String, Object> cache =
        new ConcurrentHashMap<>();

    @Override
    public String handleRequest(Object input, Context context) {
        items.add("list item");  // Thread-safe
        cache.put("key", input); // Thread-safe
        return "Success";
    }
}
```

Répertoire /tmp partagé

Le `/tmp` répertoire est partagé entre toutes les demandes simultanées dans l'environnement d'exécution. Les écritures simultanées dans le même fichier peuvent entraîner une corruption des données, par exemple si un autre processus remplace le fichier. Pour résoudre ce problème, implémentez le verrouillage des fichiers partagés ou utilisez des noms de fichiers uniques par thread ou par demande afin d'éviter les conflits. N'oubliez pas de nettoyer les fichiers inutiles pour ne pas épuiser l'espace disponible.

Logging

L'entrelacement des journaux (les entrées des journaux provenant de différentes demandes sont entrelacées dans des journaux) est normal dans les systèmes multiconcurrents.

Les fonctions utilisant des instances gérées Lambda utilisent toujours le format de journal JSON structuré introduit avec les contrôles de journalisation avancés. Ce format inclut le`requestId`, ce qui permet de corréler les entrées du journal à une seule demande. Lorsque vous utilisez l'`LambdaLogger`objet de`context.getLogger()`, `requestId` il est automatiquement inclus dans chaque entrée du journal. Pour plus d'informations, voirthe section called “Utilisation des contrôles de journalisation avancés de Lambda avec Java”.

Contexte de la requête

L'`context`objet est lié au fil de requête. L'utilisation `context.getAwsRequestId()` fournit un accès multithread à l'ID de demande pour la demande en cours.

`context.getXrayTraceId()`À utiliser pour accéder au X-Ray Trace ID. Cela fournit un accès sécurisé à l'ID de trace pour la demande en cours. Lambda ne prend pas en charge la variable d'`_X_AMZN_TRACE_ID`environnement avec les instances gérées par Lambda. Le X-Ray Trace ID est automatiquement propagé lors de l'utilisation du AWS SDK.

`com.amazonaws.services.lambda.runtime.Context.getRemainingTimeInMillis()`À utiliser pour détecter les délais d'expiration. Pour plus d’informations, consultez the section called “Gestion des erreurs et restauration”.

Si vous utilisez des threads virtuels dans votre programme ou si vous créez des threads lors de l'initialisation, vous devrez transmettre tout contexte de demande requis à ces threads.

Initialisation et arrêt

L'initialisation de la fonction a lieu une fois par environnement d'exécution. Les objets créés lors de l'initialisation sont partagés entre les threads.

Pour les fonctions Lambda avec extensions, l'environnement d'exécution émet un signal SIGTERM lors de l'arrêt. Ce signal est utilisé par les extensions pour déclencher des tâches de nettoyage, telles que le rinçage des tampons. Vous pouvez vous abonner aux événements SIGTERM pour déclencher des tâches de nettoyage des fonctions, telles que la fermeture des connexions aux bases de données. Pour en savoir plus sur le cycle de vie de l'environnement d'exécution, veuillez consulter the section called “Environnement d’exécution”.

Versions de dépendance

Les instances gérées Lambda nécessitent les versions de package minimales suivantes :

- AWS SDK pour Java 2.0 : version 2.34.0 ou ultérieure
- AWS SDK X-Ray pour Java : version 2.20.0 ou ultérieure
- AWS Distro for OpenTelemetry - Instrumentation pour Java : version 2.20.0 ou ultérieure
- Powertools pour AWS Lambda (Java) : version 2.8.0 ou ultérieure

Outils électriques pour AWS Lambda (Java)

Powertools for AWS Lambda (Java) est compatible avec les instances gérées par Lambda et fournit des utilitaires pour la journalisation, le suivi, les métriques, etc. Pour plus d'informations, consultez Powertools for AWS Lambda (Java).

Étapes suivantes

- Vérifiez le runtime Node.js pour les instances gérées Lambda
- Passez en revue le runtime Python pour les instances gérées Lambda
- Passez en revue le runtime .NET pour les instances gérées Lambda
- En savoir plus sur le dimensionnement des instances gérées Lambda

Runtime Node.js pour les instances gérées Lambda

Pour les environnements d'exécution de Node.js, les instances gérées Lambda utilisent des threads de travail `async` avec une exécution basée sur`await`/pour gérer les demandes simultanées. L'initialisation de la fonction a lieu une fois par thread de travail. Les invocations simultanées sont gérées selon deux dimensions : les threads de travail assurent le parallélisme entre vCPUs, et l'exécution asynchrone assure la simultanéité au sein de chaque thread. Chaque demande simultanée traitée par le même thread de travail partage le même objet de gestionnaire et le même état global, ce qui nécessite un traitement sécurisé dans le cadre de plusieurs demandes simultanées.

Simultanéité maximum

Le nombre maximum de demandes simultanées que Lambda envoie à chaque environnement d'exécution est contrôlé par le `PerExecutionEnvironmentMaxConcurrency` paramètre de configuration de la fonction. Il s'agit d'un paramètre facultatif, et la valeur par défaut varie en fonction du temps d'exécution. Pour les environnements d'exécution de Node.js, la valeur par défaut est de

64 requêtes simultanées par vCPU, ou vous pouvez configurer votre propre valeur. Lambda ajuste automatiquement le nombre de demandes simultanées jusqu'au maximum configuré en fonction de la capacité de chaque environnement d'exécution à absorber ces demandes.

Pour Node.js, le nombre de demandes simultanées que chaque environnement d'exécution peut traiter est déterminé par le nombre de threads de travail et par la capacité de chaque thread de travail à traiter les demandes simultanées de manière asynchrone. Le nombre par défaut de threads de travail est déterminé par le nombre de v CPUs disponibles, ou vous pouvez configurer le nombre de threads de travail en définissant la variable d'`AWS_LAMBDA_NODEJS_WORKER_COUNT`environnement. Nous recommandons d'utiliser des gestionnaires de fonctions asynchrones, car cela permet de traiter plusieurs demandes par thread de travail. Si votre gestionnaire de fonctions est synchrone, chaque thread de travail ne peut traiter qu'une seule demande à la fois.

Création de fonctions pour la multisimultanéité

Avec un gestionnaire de fonctions asynchrones, chaque utilisateur d'exécution traite plusieurs demandes simultanément. Les objets globaux seront partagés entre plusieurs demandes simultanées. Pour les objets mutables, évitez d'utiliser un état ou un usage `AsyncLocalStorage` global.

AWS Les clients du SDK sont sécurisés en mode asynchrone et ne nécessitent aucune manipulation particulière.

Exemple : État global

Le code suivant utilise un objet global qui est muté dans le gestionnaire de fonctions. Ce n'est pas sécurisé en mode asynchrone.

```
let state = {
    currentUser: null,
    requestData: null
};

export const handler = async (event, context) => {
    state.currentUser = event.userId;
    state.requestData = event.data;

    await processData(state.requestData);

    // state.currentUser might now belong to a different request
    return { user: state.currentUser };
```

```
};
```

L'initialisation de l'`state`objet dans le gestionnaire de fonctions permet d'éviter un état global partagé.

```
export const handler = async (event, context) => {
    let state = {
        currentUser: event.userId,
        requestData: event.data
    };

    await processData(state.requestData);

    return { user: state.currentUser };
};
```

Exemple : connexions à la base de données

Le code suivant utilise un objet client partagé qui est partagé entre plusieurs invocations. Selon la bibliothèque de connexions utilisée, cela peut ne pas être sécurisé en matière de simultanéité.

```
const { Client } = require('pg');

// Single connection created at init time
const client = new Client({
  host: process.env.DB_HOST,
  database: process.env.DB_NAME,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD
});

// Connect once during cold start
client.connect();

exports.handler = async (event) => {
  // Multiple parallel invocations share this single connection = BAD
  // With multi-concurrent Lambda, queries will collide
  const result = await client.query('SELECT * FROM users WHERE id = $1',
 [event.userId]);

  return {
    statusCode: 200,
```

```
    body: JSON.stringify(result.rows[0])
  };
};
```

Une approche sûre en matière de simultanéité consiste à utiliser un pool de connexions. Le pool utilise une connexion distincte pour chaque requête de base de données simultanée.

```
const { Pool } = require('pg');

// Connection pool created at init time
const pool = new Pool({
  host: process.env.DB_HOST,
  database: process.env.DB_NAME,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
  max: 20,  // Max connections in pool
  idleTimeoutMillis: 30000,
  connectionTimeoutMillis: 2000
});

exports.handler = async (event) => {
  // Pool gives each parallel invocation its own connection
  const result = await pool.query('SELECT * FROM users WHERE id = $1', [event.userId]);

  return {
    statusCode: 200,
    body: JSON.stringify(result.rows[0])
  };
};
```

Node.js 22 gestionnaires basés sur des rappels

Lorsque vous utilisez Node.js 22, vous ne pouvez pas utiliser de gestionnaire de fonctions basé sur le rappel avec les instances gérées Lambda. Les gestionnaires basés sur le rappel ne sont pris en charge que pour les fonctions Lambda (par défaut). Pour les environnements d'exécution de Node.js 24 et versions ultérieures, les gestionnaires de fonctions basés sur le rappel sont déconseillés pour les instances Lambda (par défaut) et Lambda Managed Instances.

Utilisez plutôt un gestionnaire de `async` fonctions lorsque vous utilisez des instances gérées Lambda. Pour plus d'informations, voir Définir le gestionnaire de fonctions Lambda dans le fichier Node.js.

Répertoire /tmp partagé

Le `/tmp` répertoire est partagé entre toutes les demandes simultanées dans l'environnement d'exécution. Les écritures simultanées dans le même fichier peuvent entraîner une corruption des données, par exemple si un autre processus remplace le fichier. Pour résoudre ce problème, implémentez le verrouillage des fichiers partagés ou utilisez des noms de fichiers uniques par demande afin d'éviter les conflits. N'oubliez pas de nettoyer les fichiers inutiles pour ne pas épuiser l'espace disponible.

Logging

L'entrelacement des journaux (les entrées des journaux provenant de différentes demandes sont entrelacées dans des journaux) est normal dans les systèmes multiconcurrents. Les fonctions utilisant des instances gérées Lambda utilisent toujours le format de journal JSON structuré introduit avec les contrôles de journalisation avancés. Ce format inclut le`requestId`, ce qui permet de corréler les entrées du journal à une seule demande. Lorsque vous utilisez l'`console`enregistreur, il `requestId` est automatiquement inclus dans chaque entrée du journal. Pour de plus amples informations, voirthe section called “Utilisation des contrôles de journalisation avancés de Lambda avec Node.js”.

Les bibliothèques de journalisation tierces les plus populaires, telles que Winston, prennent généralement en charge l'utilisation de la console pour la sortie des journaux.

Contexte de la requête

L'utilisation `context.awsRequestId` fournit un accès asynchrone à l'ID de demande pour la demande en cours.

`context.xRayTraceId`À utiliser pour accéder au X-Ray Trace ID. Cela fournit un accès simultané à l'ID de trace pour la demande en cours. Lambda ne prend pas en charge la variable d'`_X_AMZN_TRACE_ID`environnement avec les instances gérées par Lambda. Le X-Ray Trace ID est automatiquement propagé lors de l'utilisation du AWS SDK.

`context.getRemainingTimeInMillis()`À utiliser pour détecter les délais d'expiration. Pour plus d’informations, consultez the section called “Gestion des erreurs et restauration”.

Initialisation et arrêt

L'initialisation de la fonction a lieu une fois par thread de travail. Des entrées de journal répétées peuvent s'afficher si votre fonction émet des journaux lors de l'initialisation.

Pour les fonctions Lambda avec extensions, l'environnement d'exécution émet un signal SIGTERM lors de l'arrêt. Ce signal est utilisé par les extensions pour déclencher des tâches de nettoyage, telles que le vidage des tampons. Les fonctions Lambda (par défaut) avec extensions peuvent également s'abonner au signal SIGTERM en utilisant. `process.on()` Ceci n'est pas pris en charge pour les fonctions utilisant des instances gérées Lambda car il `process.on()` ne peut pas être utilisé avec les threads de travail. Pour en savoir plus sur le cycle de vie de l'environnement d'exécution, veuillez consulter the section called “Environnement d’exécution”.

Versions de dépendance

Les instances gérées Lambda nécessitent les versions de package minimales suivantes :

- AWS SDK pour JavaScript v3 : version 3.933.0 ou ultérieure
- AWS Kit de développement X-Ray pour Node.js : version 3.12.0 ou ultérieure
- AWS Distro pour OpenTelemetry - Instrumentation pour JavaScript : version 0.8.0 ou ultérieure
- Powertools pour AWS Lambda TypeScript () : version 2.29.0 ou ultérieure

Outils électriques pour AWS TypeScript Lambda ()

Powertools for AWS Lambda TypeScript () est compatible avec les instances gérées par Lambda et fournit des utilitaires pour la journalisation, le suivi, les métriques, etc. Pour plus d'informations, consultez Powertools for AWS Lambda TypeScript ().

Étapes suivantes

- Passez en revue l'environnement d'exécution Java pour les instances gérées Lambda
- Passez en revue le runtime Python pour les instances gérées Lambda
- Passez en revue le runtime .NET pour les instances gérées Lambda
- En savoir plus sur le dimensionnement des instances gérées Lambda

Runtime Python pour les instances gérées Lambda

Le moteur d'exécution Lambda utilise plusieurs processus Python pour gérer les requêtes simultanées. Chaque demande simultanée s'exécute dans le cadre d'un processus distinct avec son propre espace mémoire et son propre processus d'initialisation. Chaque processus traite une demande à la fois, de manière synchrone. Les processus ne partagent pas directement la mémoire,

de sorte que les variables globales, les caches au niveau du module et les objets singleton sont isolés entre les demandes simultanées.

Configuration de la simultanéité

Le nombre maximum de demandes simultanées que Lambda envoie à chaque environnement d'exécution est contrôlé par le `PerExecutionEnvironmentMaxConcurrency` paramètre de configuration de la fonction. Il s'agit d'un paramètre facultatif, et la valeur par défaut varie en fonction du temps d'exécution. Pour les environnements d'exécution Python, la valeur par défaut est de 16 requêtes simultanées par vCPU, ou vous pouvez configurer votre propre valeur. Cette valeur détermine également le nombre de processus utilisés par le moteur d'exécution Python. Lambda ajuste automatiquement le nombre de demandes simultanées jusqu'au maximum configuré en fonction de la capacité de chaque environnement d'exécution à absorber ces demandes.

> **Important**
>
> L'utilisation de la simultanéité basée sur les processus signifie que chaque processus de travail d'exécution effectue sa propre initialisation. L'utilisation totale de la mémoire est égale à la mémoire par processus multipliée par le nombre de processus simultanés. Si vous chargez des bibliothèques ou des ensembles de données volumineux et que la simultanéité est élevée, votre encombrement mémoire sera important. En fonction de votre charge de travail, vous devrez peut-être ajuster votre CPU-to-memory ratio ou utiliser un paramètre de simultanéité inférieur pour éviter de dépasser la mémoire disponible. Vous pouvez utiliser la `MemoryUtilization` métrique CloudWatch pour suivre la consommation de mémoire.

Création de fonctions pour la multisimultanéité

En raison du modèle de multisimultanéité basé sur les processus, les fonctions Lambda Managed Instances utilisant des environnements d'exécution Python n'accèdent pas simultanément aux ressources en mémoire à partir de plusieurs appels. Il n'est pas nécessaire d'appliquer des pratiques de codage pour garantir la sécurité de la simultanéité en mémoire.

Répertoire /tmp partagé

Le `/tmp` répertoire est partagé entre toutes les demandes simultanées dans l'environnement d'exécution. Les écritures simultanées dans le même fichier peuvent entraîner une corruption des données, par exemple si un autre processus remplace le fichier. Pour résoudre ce problème, implémentez le verrouillage des fichiers partagés ou utilisez des noms de fichiers uniques par

processus ou par demande afin d'éviter les conflits. N'oubliez pas de nettoyer les fichiers inutiles pour ne pas épuiser l'espace disponible.

Logging

L'entrelacement des journaux (les entrées des journaux provenant de différentes demandes sont entrelacées dans des journaux) est normal dans les systèmes multiconcurrents.

Les fonctions utilisant des instances gérées Lambda utilisent toujours le format de journal JSON structuré introduit avec les contrôles de journalisation avancés. Ce format inclut le`requestId`, ce qui permet de corréler les entrées du journal à une seule demande. Lorsque vous utilisez le `logging` module de la bibliothèque standard Python dans Lambda, `requestId` il est automatiquement inclus dans chaque entrée du journal. Pour plus d'informations, consultez la section Utilisation des commandes de journalisation avancées Lambda avec Python.

Contexte de la requête

Permet `context.aws_request_id` d'accéder à l'ID de demande pour la demande en cours.

Avec les environnements d'exécution Python, vous pouvez utiliser la variable d'`_X_AMZN_TRACE_ID`environnement pour accéder à l'ID de trace X-Ray avec les instances gérées Lambda. Le X-Ray Trace ID est automatiquement propagé lors de l'utilisation du AWS SDK.

`context.get_remaining_time_in_millis()`À utiliser pour détecter les délais d'expiration. Pour plus d'informations, consultez the section called “Gestion des erreurs et restauration”.

Initialisation et arrêt

L'initialisation de la fonction a lieu une fois par processus. Des entrées de journal répétées peuvent s'afficher si votre fonction émet des journaux lors de l'initialisation.

Pour les fonctions Lambda avec extensions, l'environnement d'exécution émet un signal SIGTERM lors de l'arrêt. Ce signal est utilisé par les extensions pour déclencher des tâches de nettoyage, telles que le rinçage des tampons. Vous pouvez vous abonner aux événements SIGTERM pour déclencher des tâches de nettoyage des fonctions, telles que la fermeture des connexions aux bases de données. Pour en savoir plus sur le cycle de vie de l'environnement d'exécution, veuillez consulter the section called “Environnement d’exécution”.

Versions de dépendance

Les instances gérées Lambda nécessitent les versions de package minimales suivantes :

- Powertools pour AWS Lambda (Python) : version 3.23.0 ou ultérieure

Powertools pour AWS Lambda (Python)

Powertools for AWS Lambda (Python) est compatible avec les instances gérées par Lambda et fournit des utilitaires pour la journalisation, le suivi, les métriques, etc. Pour plus d'informations, consultez Powertools for AWS Lambda (Python).

Étapes suivantes

- Passez en revue l'environnement d'exécution Java pour les instances gérées Lambda
- Passez en revue le runtime Node.js pour les instances gérées Lambda
- Passez en revue le runtime .NET pour les instances gérées Lambda
- En savoir plus sur le dimensionnement des instances gérées Lambda

Environnement d'exécution .NET pour les instances gérées Lambda

Pour les environnements d'exécution .NET, les instances gérées Lambda utilisent un seul processus .NET par environnement d'exécution. Plusieurs demandes simultanées sont traitées à l'aide de tâches .NET.

Configuration de la simultanéité

Le nombre maximum de demandes simultanées que Lambda envoie à chaque environnement d'exécution est contrôlé par le `PerExecutionEnvironmentMaxConcurrency` paramètre de configuration de la fonction. Il s'agit d'un paramètre facultatif, et la valeur par défaut varie en fonction du temps d'exécution. Pour les environnements d'exécution .NET, la valeur par défaut est de 32 requêtes simultanées par vCPU, ou vous pouvez configurer votre propre valeur. Lambda ajuste automatiquement le nombre de demandes simultanées jusqu'au maximum configuré en fonction de la capacité de chaque environnement d'exécution à absorber ces demandes.

Création de fonctions pour la multisimultanéité

Lorsque vous utilisez des instances gérées Lambda, vous devez appliquer les mêmes pratiques de sécurité en matière de simultanéité que dans tout autre environnement multiconcurrent. Étant donné que l'objet du gestionnaire est partagé entre toutes les tâches, tout état mutable doit être adapté aux threads. Cela inclut les collections, les connexions aux bases de données et tous les objets statiques modifiés lors du traitement des demandes.

AWS Les clients du SDK sont compatibles avec les threads et ne nécessitent aucune manipulation particulière.

Exemple : pools de connexions aux bases de données

Le code suivant utilise un objet de connexion à une base de données statique qui est partagé entre des demandes simultanées. L'`SqlConnection`objet n'est pas compatible avec les threads.

```
public class DBQueryHandler
{
    // Single connection shared across threads - NOT SAFE
    private SqlConnection connection;

    public DBQueryHandler()
    {
        connection = new SqlConnection("your-connection-string-here");
        connection.Open();
    }

    public string Handle(object input, ILambdaContext context)
    {
        using var cmd = connection.CreateCommand();
        cmd.CommandText = "SELECT ..."; // your query

        using var reader = cmd.ExecuteReader();

        ...
    }
}
```

Pour résoudre ce problème, utilisez une connexion distincte pour chaque demande, issue d'un pool de connexions. Les fournisseurs ADO.NET prennent en charge `Microsoft.Data.SqlClient` automatiquement le regroupement de connexions lorsque l'objet de connexion est ouvert.

```
public class DBQueryHandler
{
    public DBQueryHandler()
    {
    }

    public string Handle(object input, ILambdaContext context)
    {
        using var connection = new SqlConnection("your-connection-string-here");
```

```
        connection.Open();
        using var cmd = connection.CreateCommand();
        cmd.CommandText = "SELECT ..."; // your query

        using var reader = cmd.ExecuteReader();

        ...
    }
}
```

Exemple : Collections

Les collections .NET standard ne sont pas adaptées aux threads :

```
public class Handler
{
    private static List<string> items = new List<string>();
    private static Dictionary<string, object> cache = new Dictionary<string, object>();

    public string FunctionHandler(object input, ILambdaContext context)
    {
        items.Add(context.AwsRequestId);
        cache["key"] = input;

        return "Success";
    }
}
```

Utilisez les collections de l'espace de `System.Collections.Concurrent` noms pour des raisons de sécurité en matière de concurrence :

```
public class Handler
{
    private static ConcurrentBag<string> items = new ConcurrentBag<string>();
    private static ConcurrentDictionary<string, object> cache = new
 ConcurrentDictionary<string, object>();

    public string FunctionHandler(object input, ILambdaContext context)
    {
        items.Add(context.AwsRequestId);
        cache["key"] = input;

        return "Success";
```

```
    }
}
```

Répertoire /tmp partagé

Le `/tmp` répertoire est partagé entre toutes les demandes simultanées dans l'environnement d'exécution. Les écritures simultanées dans le même fichier peuvent entraîner une corruption des données, par exemple si une autre demande remplace le fichier. Pour résoudre ce problème, implémentez le verrouillage des fichiers partagés ou utilisez des noms de fichiers uniques par demande afin d'éviter les conflits. N'oubliez pas de nettoyer les fichiers inutiles pour ne pas épuiser l'espace disponible.

Logging

L'entrelacement des journaux (les entrées des journaux provenant de différentes demandes sont entrelacées dans des journaux) est normal dans les systèmes multiconcurrents. Les fonctions utilisant des instances gérées Lambda utilisent toujours le format de journal JSON structuré introduit avec les contrôles de journalisation avancés. Ce format inclut le`requestId`, ce qui permet de corréler les entrées du journal à une seule demande. Lorsque vous utilisez l'`context.Logger`objet pour générer des journaux, ceux-ci `requestId` sont automatiquement inclus dans chaque entrée de journal. Pour de plus amples informations, voirthe section called “Utilisation des contrôles de journalisation avancés de Lambda avec .NET”.

Contexte de la requête

Utilisez la `context.AwsRequestId` propriété pour accéder à l'ID de demande pour la demande en cours.

Utilisez la `context.TraceId` propriété pour accéder au X-Ray Trace ID. Cela fournit un accès simultané à l'ID de trace pour la demande en cours. Lambda ne prend pas en charge la variable d'`_X_AMZN_TRACE_ID`environnement avec les instances gérées par Lambda. Le X-Ray Trace ID est automatiquement propagé lors de l'utilisation du AWS SDK.

`ILambdaContext.RemainingTime`À utiliser pour détecter les délais d'expiration. Pour plus d’informations, consultez the section called “Gestion des erreurs et restauration”.

Initialisation et arrêt

L'initialisation de la fonction a lieu une fois par environnement d'exécution. Les objets créés lors de l'initialisation sont partagés entre les demandes.

Pour les fonctions Lambda avec extensions, l'environnement d'exécution émet un signal SIGTERM lors de l'arrêt. Ce signal est utilisé par les extensions pour déclencher des tâches de nettoyage, telles que le vidage des tampons. Vous pouvez vous abonner aux événements SIGTERM pour déclencher des tâches de nettoyage des fonctions, telles que la fermeture des connexions aux bases de données. Pour en savoir plus sur le cycle de vie de l'environnement d'exécution, veuillez consulter the section called “Environnement d’exécution”.

Versions de dépendance

Les instances gérées Lambda nécessitent les versions de package minimales suivantes :

- Amazon.Lambda.Core : version 2.7.1 ou ultérieure
- Amazon Lambda. RuntimeSupport: version 1.14.1 ou ultérieure
- OpenTelemetry.Instrumentation. AWSLambda: version 1.14.0 ou ultérieure
- AWSXRayRecorder.Core : version 2.16.0 ou ultérieure
- AWSSDK.Core : version 4.0.0.32 ou ultérieure

Outils électriques pour AWS Lambda (.NET)

Powertools for AWS Lambda (.NET) AWS et Distro OpenTelemetry for - Instrumentation DotNet for ne prennent actuellement pas en charge les instances gérées par Lambda.

Étapes suivantes

- Passez en revue l'environnement d'exécution Java pour les instances gérées Lambda
- Passez en revue le runtime Node.js pour les instances gérées Lambda
- Passez en revue le runtime Python pour les instances gérées Lambda
- En savoir plus sur le dimensionnement des instances gérées Lambda

Mise en réseau pour les instances gérées par Lambda

Lorsque vous exécutez les fonctions d'instances gérées Lambda, vous devez configurer la connectivité réseau pour permettre à vos fonctions d'accéder à des ressources en dehors du VPC. Cela inclut des AWS services tels qu'Amazon S3 et DynamoDB. La connectivité est également nécessaire pour transmettre des données de télémétrie à CloudWatch Logs and X-Ray.

Options de connectivité

Il existe trois approches principales pour configurer la connectivité VPC, chacune avec des compromis différents en termes de coût, de sécurité et de complexité.

Sous-réseau public avec passerelle Internet

Cette option utilise un sous-réseau public avec un accès direct à Internet via une passerelle Internet. Vous pouvez choisir entre IPv4 et IPv6 configurations.

IPv4 avec passerelle Internet

Pour configurer IPv4 la connectivité avec une passerelle Internet

1. Créez ou utilisez un sous-réseau public existant avec un bloc IPv4 CIDR.
2. Attachez une passerelle Internet à votre VPC.
3. Mettez à jour la table de routage pour acheminer le `0.0.0.0/0` trafic vers la passerelle Internet.
4. Assurez-vous que des IPv4 adresses publiques ou des adresses IP élastiques sont attribuées aux ressources.
5. Configurez les groupes de sécurité pour autoriser le trafic sortant sur les ports requis.

Cette configuration fournit une connectivité bidirectionnelle, autorisant à la fois les connexions sortantes depuis vos fonctions et les connexions entrantes depuis Internet.

IPv6 avec passerelle Internet

Pour configurer IPv6 la connectivité avec une passerelle Internet

1. Activez IPv6 sur votre VPC.
2. Créez ou utilisez un sous-réseau public existant auquel un bloc IPv6 CIDR est attribué.
3. Connectez une passerelle Internet à votre VPC (la même passerelle Internet peut gérer IPv4 les IPv6 deux).
4. Mettez à jour la table de routage pour acheminer le `::/0` trafic vers la passerelle Internet.
5. Vérifiez les AWS services dont vous avez besoin pour accéder à l'assistance IPv6 dans votre région.
6. Configurez les groupes de sécurité pour autoriser le trafic sortant sur les ports requis.

Cette configuration fournit une connectivité bidirectionnelle à l'aide de l' IPv6 adressage.

IPv6 avec passerelle Internet de sortie uniquement

Pour configurer IPv6 la connectivité avec une passerelle Internet de sortie uniquement

1. Activez IPv6 sur votre VPC.
2. Créez ou utilisez un sous-réseau public existant auquel un bloc IPv6 CIDR est attribué.
3. Connectez une passerelle Internet de sortie uniquement à votre VPC.
4. Mettez à jour la table de routage pour acheminer le `::/0` trafic vers la passerelle Internet de sortie uniquement.
5. Vérifiez les AWS services dont vous avez besoin pour accéder à l'assistance IPv6 dans votre région.
6. Configurez les groupes de sécurité pour autoriser le trafic sortant sur les ports requis.

Cette configuration fournit une connectivité sortante uniquement, empêchant les connexions entrantes en provenance d'Internet tout en permettant à vos fonctions d'initier des connexions sortantes.

Points de terminaison d’un VPC

Les points de terminaison VPC vous permettent de connecter en privé votre VPC aux AWS services pris en charge sans avoir besoin d'une passerelle Internet, d'un périphérique NAT, d'une connexion VPN ou d'une connexion Direct Connect. AWS Le trafic entre votre VPC et le AWS service ne quitte pas le réseau Amazon.

Pour configurer les points de terminaison VPC

1. Ouvrez la console Amazon VPC à l'adresse console.aws.amazon.com/vpc/.
2. Dans le panneau de navigation, choisissez Points de terminaison.
3. Choisissez Créer un point de terminaison.
4. Pour Service category (Catégorie de service), choisissez Services AWS .
5. Dans Nom du service, sélectionnez le point de terminaison de service dont vous avez besoin (par exemple, `com.amazonaws.region.s3` pour Amazon S3).
6. Pour VPC, sélectionnez votre VPC.

7. Pour les sous-réseaux, sélectionnez les sous-réseaux dans lesquels vous souhaitez créer des interfaces réseau de point de terminaison. Pour une haute disponibilité, sélectionnez des sous-réseaux dans plusieurs zones de disponibilité.
8. Pour Security groups (Groupes de sécurité), sélectionnez les groupes de sécurité à associer aux interfaces réseau du point de terminaison. Les groupes de sécurité doivent autoriser le trafic entrant en provenance du groupe de sécurité de votre fonction sur les ports requis.
9. Choisissez Créer un point de terminaison.

Répétez ces étapes pour chaque AWS service auquel vos fonctions doivent accéder.

Sous-réseau privé avec passerelle NAT

Cette option utilise une passerelle NAT pour fournir un accès Internet aux ressources des sous-réseaux privés tout en préservant la confidentialité des ressources.

Pour configurer un sous-réseau privé avec une passerelle NAT

1. Créez un sous-réseau public (s'il n'en existe pas déjà un) avec un bloc CIDR.
2. Attachez une passerelle Internet à votre VPC.
3. Créez une passerelle NAT dans le sous-réseau public et attribuez une adresse IP élastique.
4. Mettez à jour la table de routage du sous-réseau public pour ajouter une route : `0.0.0.0/0` → passerelle Internet.
5. Créez ou utilisez un sous-réseau privé existant avec un bloc CIDR.
6. Mettez à jour la table de routage du sous-réseau privé pour ajouter une route : `0.0.0.0/0` → passerelle NAT.
7. Configurez les groupes de sécurité pour autoriser le trafic sortant sur les ports requis.

Pour une haute disponibilité, déployez une passerelle NAT dans chaque zone de disponibilité et configurez des tables de routage par zone de disponibilité pour utiliser la passerelle NAT locale. Cela permet d'éviter les frais de transfert de données entre AZ et d'améliorer la résilience.

Choix d'une option de connectivité

Tenez compte des facteurs suivants lorsque vous choisissez une option de connectivité :

Sous-réseau public avec passerelle Internet

- Configuration la plus simple au moindre coût
- Adapté aux environnements de développement et de test
- Les ressources peuvent recevoir des connexions entrantes depuis Internet (considérations de sécurité)
- Supporte à la fois IPv4 et IPv6

Points de terminaison d'un VPC

- Sécurité maximale, le trafic reste sur le AWS réseau
- Latence plus faible par rapport au routage Internet
- Recommandé pour les environnements de production soumis à des exigences de sécurité strictes
- Coût plus élevé par point de terminaison, par zone de disponibilité et par Go traité
- Nécessite un point de terminaison dans chaque zone de disponibilité pour une haute disponibilité

Sous-réseau privé avec passerelle NAT

- Les ressources restent privées sans accès Internet entrant
- Modèle d'architecture d'entreprise standard
- Supporte tout le trafic IPv4 Internet
- Coût modéré avec frais horaires de passerelle NAT et frais de traitement des données
- Supports IPv4 uniquement

Étapes suivantes

- En savoir plus sur les fournisseurs de capacité pour les instances gérées Lambda
- Comprendre le dimensionnement pour les instances gérées Lambda
- Consultez les guides spécifiques à l'exécution pour Java, Node.js et Python
- Comprendre la sécurité et les autorisations pour les instances gérées Lambda

Surveillance des instances gérées par Lambda

Vous pouvez surveiller les instances gérées Lambda à l'aide CloudWatch de métriques. Lambda publie automatiquement des métriques pour vous aider CloudWatch à surveiller l'utilisation des ressources, à suivre les coûts et à optimiser les performances.

Métriques disponibles

Les instances gérées Lambda fournissent des métriques à deux niveaux : au niveau du fournisseur de capacité et au niveau de l'environnement d'exécution.

Indicateurs au niveau du fournisseur de capacité

Les indicateurs au niveau du fournisseur de capacité fournissent une visibilité sur l'utilisation globale des ressources dans l'ensemble de vos instances. Ces mesures utilisent les dimensions suivantes :

- CapacityProviderName- Le nom de votre fournisseur de capacité
- InstanceType- Le type d' EC2 instance

Indicateurs d'utilisation des ressources :

- CPUUtilization- Le pourcentage d'utilisation du processeur entre les instances du fournisseur de capacité
- MemoryUtilization- Le pourcentage d'utilisation de la mémoire entre les instances du fournisseur de capacité
- NetworkOut- Trafic réseau envoyé via l'ENI du client (en octets)
- NetworkIn- Trafic réseau reçu via l'ENI du client (en octets)
- DiskReadBytes- Lire le trafic depuis le stockage local entre les instances (en octets)
- DiskWriteBytes- Écrire le trafic vers le stockage local entre les instances (en octets)

Indicateurs de capacité :

- v CPUAvailable - La quantité de vCPU disponible sur les instances pour l'allocation (en nombre)
- MemoryAvailable- La quantité de mémoire disponible sur les instances pour l'allocation (en octets)
- v CPUAllocated - La quantité de vCPU allouée aux instances pour les environnements d'exécution (en nombre)

- MemoryAllocated- La quantité de mémoire allouée aux instances pour les environnements d'exécution (en octets)

Mesures au niveau de l'environnement d'exécution

Les métriques au niveau de l'environnement d'exécution fournissent une visibilité sur l'utilisation des ressources et la simultanéité des fonctions individuelles. Ces mesures utilisent les dimensions suivantes :

- CapacityProviderName- Le nom de votre fournisseur de capacité
- FunctionName- Le nom de votre fonction Lambda

Métriques de l'environnement d'exécution disponibles :

- ExecutionEnvironmentConcurrency- La simultanéité maximale sur une période d'échantillonnage de 5 minutes
- ExecutionEnvironmentConcurrencyLimit- La limite maximale de simultanéité par environnement d'exécution
- ExecutionEnvironmentCPUUtilization- Le pourcentage d'utilisation du processeur pour les environnements d'exécution de la fonction
- ExecutionEnvironmentMemoryUtilization- Le pourcentage d'utilisation de la mémoire pour les environnements d'exécution de la fonction

Fréquence métrique et rétention

Les métriques des instances gérées Lambda sont publiées à intervalles de 5 minutes et conservées pendant 15 mois.

Afficher les métriques dans CloudWatch

Pour consulter les métriques des instances gérées Lambda dans la console CloudWatch

1. Ouvrez la CloudWatch console à l'adresse console.aws.amazon.com/cloudwatch/.
2. Dans le panneau de navigation, sélectionnez Métriques.
3. Dans l'onglet Toutes les métriques, choisissez AWS/Lambda.
4. Choisissez la dimension métrique que vous souhaitez afficher :

- Pour les mesures au niveau du fournisseur de capacité, filtrez par CapacityProviderNameet InstanceType
- Pour les métriques au niveau de l'environnement d'exécution, filtrez par CapacityProviderNameet FunctionName

5. Sélectionnez les indicateurs que vous souhaitez surveiller.

Utiliser des métriques pour optimiser les performances

Surveillez l'utilisation du processeur et de la mémoire pour déterminer si vos fonctions sont correctement dimensionnées. Une utilisation élevée peut indiquer le besoin de types d'instances plus grands ou d'une augmentation de l'allocation de mémoire aux fonctions. Suivez les indicateurs de simultanéité pour comprendre le comportement de mise à l'échelle et identifier les ralentissements potentiels.

Surveillez les indicateurs de capacité pour vous assurer que des ressources suffisantes sont disponibles pour vos charges de travail. Le v CPUAvailable et MemoryAvailableles métriques vous aident à comprendre la capacité restante de vos instances.

Étapes suivantes

- En savoir plus sur le dimensionnement des instances gérées Lambda
- Consultez les guides spécifiques à l'exécution pour Java, Node.js et Python
- Configurez la connectivité VPC pour vos fournisseurs de capacité
- Comprendre la sécurité et les autorisations pour les instances gérées Lambda

Quotas d'instances gérées Lambda

Cette page décrit les quotas de service pour les instances gérées AWS Lambda. Ces quotas sont distincts des quotas AWS Lambda (par défaut). Certains quotas peuvent être augmentés sur demande.

Quotas de demandes d'API Lambda

Ces quotas contrôlent le taux auquel vous pouvez effectuer des appels d'API pour gérer les fournisseurs de capacité des instances gérées Lambda. Les limites de débit des API de lecture et

d'écriture s'appliquent à toutes les opérations des fournisseurs de capacité combinées, y compris la création, la mise à jour, la description et la suppression de fournisseurs de capacité.

Ressource	Quota
Le taux combiné maximal (demandes par seconde) pour tous les fournisseurs de capacité lus APIs	15 requêtes par seconde. Il ne peut pas être augmenté.
Le taux combiné maximal (demandes par seconde) pour toutes les capacités d'écriture du fournisseur APIs	1 demande par seconde. Il ne peut pas être augmenté.

Quotas de ressources pour les instances gérées Lambda

Ces quotas définissent les limites des ressources principales des instances gérées Lambda au sein de votre AWS compte. Ils régissent le nombre de fournisseurs de capacité que vous pouvez créer et le nombre de versions de fonctions qui peuvent être associées à chaque fournisseur de capacité.

Ressource	Quota
Fournisseurs de capacité	1 000. Le nombre maximum de fournisseurs de capacité créés dans un compte.
Versions de fonctions par fournisseur de capacité	100. Le nombre maximum de versions de fonctions par fournisseur de capacité. Il ne peut pas être augmenté.

Quotas de mappage de la source

Ces quotas contrôlent le débit et les limites de configuration pour le traitement des événements provenant de divers AWS services sur les instances gérées Lambda. Les limites de débit garantissent des performances prévisibles, tandis que les limites du nombre de mappages contribuent à maintenir la stabilité du service. Les mappages de sources d'événements sur les instances gérées Lambda prennent en charge Amazon SQS, DynamoDB Streams, Amazon Kinesis Data Streams, Amazon MSK et Apache Kafka autogéré en tant que sources d'événements.

Ressource	Quota
Débit de mappage des sources d'événements SQS standard sur les instances gérées Lambda	5 Mo par seconde. Il ne peut pas être augmenté.
Débit de mappage des sources d'événements Kafka standard sur les instances gérées Lambda	1 Mo par seconde. Il ne peut pas être augmenté.
Mappages de sources d'événements Kafka standard sur les instances gérées Lambda	100 mappages de sources d'événements. Il ne peut pas être augmenté.
Débit de mappage des sources d'événements Kinesis sur les instances gérées Lambda	25 Mo par seconde. Peut être augmenté.
Débit de mappage des sources d'événements DynamoDB sur les instances gérées Lambda	10 Mo par seconde. Peut être augmenté.
Invoquer le débit des demandes pour les appels asynchrones sur les instances gérées Lambda	5 Mo par seconde. Peut être augmenté.

Demande d’augmentation de quota

Pour les quotas qui peuvent être augmentés, vous pouvez demander une augmentation via la console Service Quotas.

Pour demander une augmentation de quota

1. Ouvrez la console Service Quotas à l'adresse console.aws.amazon.com/servicequotas/.
2. Dans le panneau de navigation, choisissez Services AWS .
3. Choisissez AWS Lambda.
4. Sélectionnez le quota que vous souhaitez augmenter.
5. Choisissez Request quota increase (Demander une augmentation de quota).
6. Entrez la nouvelle valeur du quota et justifiez l'augmentation.
7. Cliquez sur Demander.

Étapes suivantes

- En savoir plus sur les fournisseurs de capacité pour les instances gérées Lambda
- Comprendre le dimensionnement pour les instances gérées Lambda
- Consultez les guides spécifiques à l'exécution pour Java, Node.js et Python
- Configurez la connectivité VPC pour vos fournisseurs de capacité

Bonnes pratiques pour les instances gérées Lambda

Configuration du fournisseur de capacité

Séparez les fournisseurs de capacité par niveau de confiance. Créez différents fournisseurs de capacité pour les charges de travail présentant des exigences de sécurité différentes. Toutes les fonctions assignées au même fournisseur de capacité doivent être mutuellement fiables, car les fournisseurs de capacité servent de limite de sécurité.

Utilisez des noms descriptifs. Nommez les fournisseurs de capacité pour indiquer clairement leur utilisation prévue et leur niveau de confiance (par exemple`production-trusted`,`dev-sandbox`). Cela permet aux équipes de comprendre l'objectif et le niveau de sécurité de chaque fournisseur de capacité.

Utilisez plusieurs zones de disponibilité. Spécifiez des sous-réseaux dans plusieurs zones de disponibilité lors de la création de fournisseurs de capacité. Lambda lance trois instances par défaut pour la résilience AZ, garantissant ainsi la haute disponibilité de vos fonctions.

Sélection du type d'instance

Laissez Lambda choisir les types d'instances. Par défaut, Lambda choisit les types d'instances les mieux adaptés à votre charge de travail. Nous vous recommandons de laisser les instances gérées Lambda choisir les types d'instances à votre place, car la restriction du nombre de types d'instances possibles peut entraîner une baisse de disponibilité.

Spécifiez les types d'instances pour des besoins spécifiques. Si vous avez des exigences matérielles spécifiques, définissez les types d'instances autorisés dans une liste d'instances compatibles. Par exemple :

- Pour les applications nécessitant une bande passante réseau élevée, sélectionnez plusieurs types d'instances n

- Pour les environnements de test ou de développement soumis à des contraintes de coût, choisissez des types d'instances plus petits tels que m7a.large

Configuration de la fonction

Choisissez les paramètres de mémoire et de vCPU appropriés. Sélectionnez des configurations de mémoire et de vCPU qui prennent en charge les exécutions simultanées de votre fonction. La taille de fonction minimale prise en charge est de 2 Go et 1 vCPU.

- Pour les applications Python, choisissez un ratio mémoire/v plus élevé CPUs (par exemple 4 pour 1 ou 8 pour 1) en raison de la façon dont Python gère la multisimultanéité
- Pour les opérations gourmandes en ressources CPU ou les fonctions qui effectuent peu d'E/S, choisissez plusieurs vCPU
- Pour les applications gourmandes en E/S, telles que les services Web ou les tâches par lots, la multisimultanéité offre le plus d'avantages

Configurez la simultanéité maximale de manière appropriée. Lambda choisit des valeurs par défaut raisonnables pour une simultanéité maximale qui équilibre la consommation de ressources et le débit. Ajustez ce paramètre en fonction de l'utilisation des ressources de votre fonction :

- Augmentez le maximum de simultanéité (jusqu'à 64 par vCPU) si vos invocations de fonctions utilisent très peu de CPU
- Diminuez la simultanéité maximale si votre application consomme une grande quantité de mémoire et très peu de processeur

Notez que les environnements d'exécution présentant une très faible simultanéité peuvent être confrontés à des ralentissements et à des difficultés de mise à l'échelle.

Configuration de la mise à l'échelle

Définissez une utilisation cible appropriée des ressources. Par défaut, Lambda conserve une marge de manœuvre suffisante pour que votre trafic double en 5 minutes sans limitation. Ajustez cela en fonction des caractéristiques de votre charge de travail :

- Pour des charges de travail très stables ou des applications non sensibles aux limitations, fixez l'objectif à un niveau élevé afin d'augmenter le taux d'utilisation et de réduire les coûts

- Pour les charges de travail susceptibles de générer des pics de trafic, définissez des objectifs de ressources à un niveau bas afin de conserver une marge de manœuvre supplémentaire

Planifiez la croissance du trafic. Si votre trafic double plus que dans les 5 minutes, vous risquez de constater des ralentissements à mesure que Lambda augmente le nombre d'instances et d'environnements d'exécution. Concevez votre application de manière à gérer les ralentissements potentiels pendant les périodes de mise à l'échelle rapide.

Sécurité

Appliquez le moindre privilège pour PassCapacityProvider les autorisations. N'accordez `lambda:PassCapacityProvider` des autorisations qu'aux fournisseurs de capacité nécessaires. Utilisez les autorisations au niveau des ressources pour limiter les fournisseurs de capacité que les utilisateurs peuvent affecter aux fonctions.

Surveillez l'utilisation des fournisseurs de capacité. AWS CloudTrail À utiliser pour surveiller les affectations des fournisseurs de capacité et les modèles d'accès. Cela permet d'identifier les tentatives d'accès non autorisées et de garantir le respect des politiques de sécurité.

Séparez les charges de travail non fiables. Ne vous fiez pas aux conteneurs pour isoler la sécurité entre les charges de travail non fiables. Utilisez différents fournisseurs de capacité pour séparer les charges de travail qui ne sont pas mutuellement fiables.

Optimisation des coûts

Tirez parti EC2 des options de tarification. Profitez des EC2 Savings Plans et des instances réservées pour réduire les coûts. Ces options de tarification s'appliquent au EC2 calcul sous-jacent (les frais de gestion de 15 % ne sont pas réduits).

Optimisez les charges de travail en régime permanent. Les instances gérées Lambda sont particulièrement adaptées aux fonctions en régime permanent avec un volume de trafic élevé prévisible. Pour les modèles de trafic en rafale, Lambda (par défaut) peut être plus rentable.

Surveillez l'utilisation des ressources. Suivez CloudWatch les métriques pour comprendre l'utilisation du processeur et de la mémoire. Ajustez l'allocation de mémoire des fonctions et la sélection du type d'instance en fonction des modèles d'utilisation réels afin d'optimiser les coûts.

Surveillance et observabilité

Surveillez les indicateurs des fournisseurs de capacité. Suivez les indicateurs au niveau des fournisseurs de capacité CPUUtilization, y compris MemoryUtilization, vCPUAvailable, et MemoryAvailable pour vous assurer que des ressources suffisantes sont disponibles pour vos charges de travail.

Surveillez les métriques de l'environnement d'exécution. Suivez les indicateurs au niveau de l'environnement d'exécution, notamment ExecutionEnvironmentConcurrency ExecutionEnvironmentConcurrencyLimit pour comprendre le comportement de dimensionnement et identifier les ralentissements potentiels.

Configurez des CloudWatch alarmes. Créez des CloudWatch alarmes pour les indicateurs clés afin d'identifier les problèmes de manière proactive :

- Utilisation élevée du processeur ou de la mémoire
- Faible capacité disponible
- À l'approche des limites de simultanéité

Considérations spécifiques à l'exécution

Suivez les meilleures pratiques spécifiques à l'environnement d'exécution. Chaque environnement d'exécution gère différemment la multisimultanéité. Consultez les guides spécifiques à l'environnement d'exécution pour obtenir des recommandations détaillées :

- Java : utilisez des collections adaptées aux threads et `ThreadLocal` pour un état `AtomicInteger` spécifique à la demande
- Node.js : à utiliser InvokeStore pour tous les états spécifiques à la demande et à éviter les variables globales
- Python : utilisez des noms de fichiers uniques dans les requêtes IDs et envisagez `/tmp` une isolation de la mémoire basée sur les processus

Testez les problèmes de sécurité et de simultanéité des threads. Avant le déploiement en production, testez soigneusement vos fonctions pour détecter les problèmes de sécurité des threads, les conditions de course et l'isolation correcte des états sous charge simultanée.

Étapes suivantes

- En savoir plus sur les fournisseurs de capacité pour les instances gérées Lambda
- Comprendre le dimensionnement pour les instances gérées Lambda
- Consultez les guides spécifiques à l'exécution pour Java, Node.js et Python
- Configurez la connectivité VPC pour vos fournisseurs de capacité
- Surveillez les instances gérées Lambda à l'aide de métriques CloudWatch

Résolution des problèmes liés aux instances gérées Lambda

Problèmes de régulation et de mise à l'échelle

Taux d'erreur élevés lors de la mise à l'échelle

Problème : vous rencontrez des erreurs de régulation (HTTP 429) lorsque le trafic augmente rapidement.

Cause : Les instances gérées Lambda évoluent de manière asynchrone en fonction de l'utilisation des ressources du processeur et de la saturation en multisimultanéité. Si votre trafic double plus que dans les 5 minutes, vous risquez de rencontrer des difficultés à mesure que Lambda augmente le volume des instances et des environnements d'exécution pour répondre à la demande.

Solution :

- Ajustez l'utilisation des ressources cibles : si votre charge de travail présente des modèles de trafic prévisibles, définissez une utilisation des ressources cible inférieure afin de conserver une marge de manœuvre supplémentaire en cas de pics de trafic.
- Capacité de préchauffage : pour les augmentations de trafic prévues, augmentez progressivement le trafic sur une plus longue période pour permettre à l'évolution de suivre le rythme.
- Surveillez les mesures de dimensionnement : suivez les mesures d'erreur d'accélération pour comprendre la raison des problèmes liés aux accélérateurs et à la mise à l'échelle de la capacité.
- Vérifiez la configuration des fonctions : assurez-vous que la mémoire des fonctions et les paramètres du vCPU prennent en charge les exécutions simultanées multiples. Augmentez la mémoire des fonctions ou l'allocation de vCPU si nécessaire.

Diminution lente

Problème : les instances mettent beaucoup de temps à être réduites lorsque le trafic diminue.

Cause : les instances gérées Lambda diminuent progressivement pour maintenir la disponibilité et éviter les changements rapides de capacité susceptibles d'avoir un impact sur les performances.

Solution :

Ce comportement est normal. Lambda réduit les instances de manière conservatrice pour garantir la stabilité. Surveillez vos CloudWatch indicateurs pour suivre le nombre d'instances en cours d'exécution.

Problèmes de simultanéité

Environnements d'exécution présentant de faibles limites d'expérience en matière de simultanéité

Problème : vos fonctions sont limitées malgré la capacité disponible.

Cause : les environnements d'exécution présentant une simultanéité maximale très faible peuvent avoir des difficultés à évoluer efficacement. Les instances gérées Lambda sont conçues pour les applications multiconcurrentes.

Solution :

- Augmenter la simultanéité maximale : si vos appels de fonctions utilisent très peu de CPU, augmentez le paramètre de simultanéité maximale à 64 par vCPU.
- Optimisation du code de fonction : passez en revue votre code de fonction pour réduire la consommation du processeur par appel, permettant ainsi une plus grande simultanéité.
- Ajustez la mémoire des fonctions et le vCPU : assurez-vous que votre fonction dispose de suffisamment de ressources pour gérer plusieurs appels simultanés.

Problèmes de sécurité des threads (environnement d'exécution Java)

Problème : votre fonction Java produit des résultats incorrects ou est confrontée à des conditions de course sous charge.

Cause : plusieurs threads exécutent la méthode du gestionnaire simultanément, et l'état partagé n'est pas adapté aux threads.

Solution :

- Utilisez `AtomicInteger` ou `AtomicLong` pour des compteurs au lieu de types primitifs
- Remplacez `HashMap` par `ConcurrentHashMap`.
- Utiliser `Collections.synchronizedList()` pour emballer `ArrayList`
- À utiliser `ThreadLocal` pour un état spécifique à la demande
- Trace d'accès IDs à partir de l'objet Lambda Context, et non des variables d'environnement

Pour obtenir des instructions détaillées, consultez la documentation relative à l'environnement d'exécution Java pour les instances gérées Lambda.

Problèmes d'isolation de l'état (environnement d'exécution de Node.js)

Problème : votre fonction Node.js renvoie des données provenant de différentes requêtes ou est corrompue.

Cause : les variables globales sont partagées entre les appels simultanés sur le même thread de travail. Lorsque les opérations asynchrones donnent le contrôle, d'autres invocations peuvent modifier l'état partagé.

Solution :

- Installation et utilisation `@aws/lambda-invoke-store` pour tous les états spécifiques à la demande
- Remplacez les variables globales par `InvokeStore.set()` et `InvokeStore.get()`
- Utiliser des noms de fichiers uniques dans `/tmp` la demande IDs
- Accédez au suivi IDs en utilisant `InvokeStore.getXRayTraceId()` plutôt que des variables d'environnement

Pour obtenir des instructions détaillées, consultez la documentation du runtime Node.js pour les instances gérées Lambda.

Conflits de fichiers (environnement d'exécution Python)

Problème : votre fonction Python lit des données incorrectes dans les fichiers`/tmp`.

Cause : plusieurs processus partagent le `/tmp` répertoire. Les écritures simultanées dans le même fichier peuvent entraîner une corruption des données.

Solution :

- Utilisez des noms de fichiers uniques avec la demande IDs : `/tmp/request_{context.request_id}.txt`
- Utiliser le verrouillage de fichiers avec `fcntl.flock()` pour les fichiers partagés
- Nettoyez les fichiers temporaires `os.remove()` après utilisation

Pour obtenir des instructions détaillées, consultez la documentation relative à l'environnement d'exécution Python pour les instances gérées Lambda.

Problèmes de performance

Utilisation élevée de la mémoire

Problème : vos fonctions sont confrontées à une utilisation élevée de la mémoire ou à out-of-memory des erreurs.

Cause : Chaque demande simultanée en Python s'exécute dans un processus distinct avec son propre espace mémoire. L'utilisation totale de la mémoire est égale à la mémoire par processus multipliée par les processus simultanés.

Solution :

- Surveillez la `MemoryUtilization` métrique dans CloudWatch
- Réduisez le `MaxConcurrency` paramètre si l'utilisation de la mémoire approche de la limite de mémoire de la fonction
- Augmenter l'allocation de mémoire des fonctions pour permettre une plus grande simultanéité
- Optimisez l'utilisation de la mémoire en chargeant les données à la demande plutôt que lors de l'initialisation

Performances incohérentes

Problème : les performances des fonctions varient considérablement d'un appel à l'autre.

Cause : Lambda peut sélectionner différents types d'instances en fonction de la disponibilité, ou des fonctions peuvent être exécutées sur des instances dont la disponibilité des ressources varie.

Solution :

- Spécifiez les types d'instances autorisés : si vous avez des exigences de performances spécifiques, configurez les types d'instances autorisés dans votre fournisseur de capacité afin de limiter les types d'instances que Lambda peut sélectionner.
- Surveillez les métriques au niveau de l'instance : suivez `CPUUtilization` et `MemoryUtilization` au niveau du fournisseur de capacité pour identifier les contraintes en matière de ressources.
- Passez en revue les indicateurs de capacité : vérifiez `vCPUAvailable` et `MemoryAvailable` assurez-vous que des ressources suffisantes sont disponibles sur vos instances.

Problèmes liés aux fournisseurs de capacité

La version de la fonction ne devient pas ACTIVE

Problème : la version de votre fonction reste en attente après la publication.

Cause : Lambda lance des instances gérées et démarre des environnements d'exécution. Ce processus prend du temps, en particulier pour la première version fonctionnelle d'un nouveau fournisseur de capacité.

Solution :

Attendez que Lambda termine le processus d'initialisation. Lambda lance trois instances par défaut pour la résilience AZ et démarre trois environnements d'exécution avant de marquer la version ACTIVE de votre fonction. Cela prend généralement plusieurs minutes.

Impossible de supprimer le fournisseur de capacité

Problème : vous recevez un message d'erreur lorsque vous tentez de supprimer un fournisseur de capacité.

Cause : vous ne pouvez pas supprimer un fournisseur de capacité auquel des versions de fonctions sont associées.

Solution :

1. Identifiez toutes les versions de fonctions à l'aide du fournisseur de capacité avec l'`ListFunctionVersionsByCapacityProvider`API.

2. Supprimez ou mettez à jour ces versions de fonction pour supprimer l'association du fournisseur de capacité.
3. Réessayez de supprimer le fournisseur de capacité.

Messages d'erreur génériques lors de la publication des fonctions

Problème : vous rencontrez des messages d'erreur génériques tels que « Une erreur interne s'est produite lors de la publication » lors de la publication de fonctions.

Solution :

- Vérifiez les autorisations IAM : assurez-vous d'avoir l'`lambda:PassCapacityProvider`autorisation du fournisseur de capacité que vous essayez d'utiliser.
- Vérifiez la configuration du fournisseur de capacité : vérifiez que votre fournisseur de capacité est à l'état ACTIF à l'aide de l'`GetCapacityProvider`API.
- Vérifiez la configuration du VPC : assurez-vous que les sous-réseaux et les groupes de sécurité spécifiés dans votre fournisseur de capacité sont correctement configurés et accessibles.
- Vérifier les AWS CloudTrail journaux : consultez CloudTrail les journaux pour obtenir des informations détaillées sur les erreurs relatives à l'échec de l'opération.

Problèmes de surveillance et d'observabilité

CloudWatch Métriques manquantes

Problème : vous ne voyez pas les indicateurs attendus CloudWatch pour votre fournisseur de capacité ou vos fonctions.

Cause : Les métriques sont publiées à intervalles de 5 minutes. Il est possible que les indicateurs ne soient pas disponibles immédiatement pour les nouveaux fournisseurs de capacités ou les nouvelles fonctions.

Solution :

Attendez au moins 5 à 10 minutes après la publication d'une version de fonction avant de vous attendre à ce que les métriques apparaissent dans CloudWatch. Vérifiez que vous recherchez le bon espace de noms (`AWS/Lambda`) et les bonnes dimensions (`CapacityProviderName`, `FunctionName`, ou`InstanceType`).

Impossible de trouver les CloudWatch journaux

Problème : votre fonction s'exécute correctement, mais vous ne trouvez pas les journaux dans les CloudWatch journaux.

Cause : les instances gérées Lambda s'exécutent dans votre VPC et nécessitent une connectivité réseau pour envoyer les journaux aux journaux. CloudWatch Sans une configuration de connectivité VPC appropriée, vos fonctions ne peuvent pas atteindre le point de terminaison du service CloudWatch Logs.

Solution :

Configurez la connectivité VPC pour permettre à vos fonctions d'envoyer des journaux à CloudWatch Logs. Trois possibilités s'offrent à vous :

Option 1 : point de terminaison VPC pour les CloudWatch journaux (recommandé pour la production)

1. Ouvrez la console Amazon VPC à l'adresse console.aws.amazon.com/vpc/.
2. Dans le panneau de navigation, choisissez Points de terminaison.
3. Choisissez Créer un point de terminaison.
4. Pour Service category (Catégorie de service), choisissez Services AWS .
5. Pour le nom du service, sélectionnez `com.amazonaws.region.logs` (remplacez `region` par votre AWS région).
6. Pour le VPC, sélectionnez le VPC utilisé par votre fournisseur de capacité.
7. Pour les sous-réseaux, sélectionnez les sous-réseaux dans lesquels vous souhaitez créer des interfaces réseau de point de terminaison. Pour une haute disponibilité, sélectionnez des sous-réseaux dans plusieurs zones de disponibilité.
8. Pour les groupes de sécurité, sélectionnez les groupes de sécurité qui autorisent le trafic HTTPS entrant (port 443) depuis le groupe de sécurité de votre fonction.
9. Activez le DNS privé pour le point de terminaison.
10. Choisissez Créer un point de terminaison.

Option 2 : sous-réseau public avec passerelle Internet

Si votre fournisseur de capacité utilise des sous-réseaux publics, assurez-vous que :

1. Une passerelle Internet est attachée à votre VPC

2. La table de routage achemine le `0.0.0.0/0` trafic vers la passerelle Internet
3. Les groupes de sécurité autorisent le trafic HTTPS sortant sur le port 443

Option 3 : sous-réseau privé avec passerelle NAT

Si votre fournisseur de capacité utilise des sous-réseaux privés, assurez-vous que :

1. Une passerelle NAT existe dans un sous-réseau public
2. La table de routage du sous-réseau privé achemine le `0.0.0.0/0` trafic vers la passerelle NAT
3. La table de routage du sous-réseau public achemine `0.0.0.0/0` le trafic vers une passerelle Internet
4. Les groupes de sécurité autorisent le trafic HTTPS sortant sur le port 443

Pour obtenir des instructions détaillées sur les options de connectivité VPC, consultez la section Connectivité VPC pour les instances gérées Lambda.

Difficulté à corréler les journaux à partir de demandes simultanées

Problème : les journaux des différentes demandes sont entrelacés, ce qui rend difficile le suivi des demandes individuelles.

Cause : L'entrelacement des journaux est un comportement normal et normal dans les systèmes multiconcurrents.

Solution :

- Utiliser la journalisation structurée au format JSON : inclure l'ID de demande dans toutes les instructions du journal
- Java : utilisez Log4j avec `ThreadContext` pour inclure automatiquement l'ID de demande
- Node.js : à utiliser `console.log()` avec le formatage JSON et à inclure `InvokeStore.getRequestId()`
- Python : utilisez le module de journalisation standard avec le formatage JSON et incluez `context.request_id`

Pour obtenir des instructions détaillées, consultez les pages de documentation spécifiques à l'environnement d'exécution.

Aide supplémentaire

Si vous continuez à rencontrer des problèmes après avoir essayé ces solutions :

1. Révision CloudWatch des indicateurs : vérifiez les indicateurs du fournisseur de capacité et de l'environnement d'exécution pour identifier les contraintes de ressources ou les problèmes de dimensionnement.
2. Vérifier les AWS CloudTrail journaux : consultez CloudTrail les journaux pour obtenir des informations détaillées sur les appels d'API et les erreurs.
3. Contacter le AWS support : si vous ne parvenez pas à résoudre le problème, contactez le AWS support en fournissant des informations sur la configuration de votre fournisseur de capacité, la configuration des fonctions et les messages d'erreur spécifiques que vous rencontrez.

Étapes suivantes

- En savoir plus sur les fournisseurs de capacité pour les instances gérées Lambda
- Comprendre le dimensionnement pour les instances gérées Lambda
- Consultez les guides spécifiques à l'exécution pour Java, Node.js et Python
- Surveillez les instances gérées Lambda à l'aide de métriques CloudWatch
- Passez en revue les meilleures pratiques pour les instances gérées Lambda

Environnements d’exécution (runtimes) Lambda

Lambda prend en charge plusieurs langages via l’utilisation d’environnements d’exécution. Une exécution fournit un environnement spécifique au langage qui relaie les événements d’invocation, les informations de contexte et les réponses entre Lambda et la fonction. Vous pouvez utiliser les runtimes que Lambda fournit, ou créer vos propres runtimes.

Lambda est indépendant de votre choix d’environnement d’exécution. Pour les fonctions simples, les langages interprétés tels que Python et Node.js offrent les performances les plus rapides. Pour les fonctions nécessitant des calculs plus complexes, les langages compilés tels que Java sont souvent plus lents à initialiser, mais s’exécutent rapidement dans le gestionnaire Lambda. Le choix de l’environnement d’exécution dépend également des préférences du développeur et de sa connaissance du langage.

Chaque version majeure du langage de programmation possède un environnement d’exécution distinct, avec un identifiant de l’environnement d’exécution unique, tel que `nodejs24.x` ou `python3.14`. Pour configurer une fonction afin d’utiliser une nouvelle version majeure du langage, vous devez modifier l’identifiant d’exécution. Comme il AWS Lambda n'est pas possible de garantir la rétrocompatibilité entre les versions principales, il s'agit d'une opération pilotée par le client.

Pour une fonction définie en tant qu’image de conteneur, vous choisissez une exécution et la distribution Linux lorsque vous créez l’image de conteneur. Pour modifier l’environnement d’exécution, vous créez une image de conteneur.

Lorsque vous utilisez une archive de fichiers .zip pour le package de déploiement, vous choisissez un environnement d’exécution lorsque vous créez la fonction. Pour modifier l’environnement d’exécution, vous pouvez mettre à jour la configuration de votre fonction. L’environnement d’exécution est associé à l’une des distributions Amazon Linux. L’environnement d’exécution sous-jacent fournit des bibliothèques et des variables d’environnement supplémentaires auxquelles vous pouvez accéder depuis le code de votre fonction.

Lambda invoque votre fonction dans un environnement d’exécution. L’environnement d’exécution fournit un environnement d’environnement d’exécution sécurisé et isolé qui gère les ressources nécessaires à l’exécution de votre fonction. Lambda réutilise l’environnement d’exécution à partir d’une invocation antérieure dans le cas où il y en a un disponible, ou il peut en créer un nouveau.

Pour utiliser d'autres langages dans Lambda, tels que Go ou Rust, utilisez un environnement d'exécution uniquement pour le système d'exploitation. L’environnement d’exécution de Lambda fournit une interface d’environnement d’exécution pour obtenir des événements d’invocations

et envoyer des réponses. Vous pouvez déployer les autres langages en implémentant un environnement d'exécution en association avec le code de votre fonction, ou dans une couche.

Environnements d'exécution pris en charge

Le tableau suivant répertorie les durées d'exécution Lambda prises en charge et les dates d'obsolescence prévues. Une fois qu'un environnement d'exécution est obsolète, vous pouvez toujours créer et mettre à jour des fonctions pendant une période limitée. Pour de plus amples informations, veuillez consulter the section called "Environnement d'exécution utilisé après la date d'obsolescence". Le tableau fournit les dates actuellement prévues pour la dépréciation de l'exécution sur la base de notre the section called "politique d'obsolescence de l'exécution". Ces dates sont fournies à des fins de planification et sont susceptibles d'être modifiées.

> **Important**
>
> La fin de vie d'Amazon Linux 2 est prévue pour le 30 juin 2026. Les environnements d'exécution Lambda et les images de base de conteneur pour Java 8 (AL2), Java 11, Java 17, Python 3.10, Python 3.11 et provided.al2 continueront de recevoir des correctifs pour des problèmes de sécurité critiques et certains problèmes de sécurité importants d'Amazon Linux 2, en plus des correctifs d'exécution du langage, jusqu'aux dates d'obsolescence indiquées dans le tableau ci-dessous.
>
> Nous recommandons aux clients de passer à un environnement d'exécution basé sur Amazon Linux 2023 dès que possible. Pour les clients effectuant une mise à niveau vers Java 21 ou Java 25, vous pouvez utiliser la fonction AWS Transform personnalisée pour les aider à effectuer ces mises à niveau. Pour les clients qui ne sont pas en mesure de mettre à niveau leur version de Java, nous prévoyons de publier des environnements d'exécution basés sur Amazon Linux 2023 pour Java 8, Java 11 et Java 17 avant la fin du deuxième trimestre 2026.

Name	Identifiant	Système d'exploitation	Date d'obsolescence	Créer la fonction de blocage	Mettre à jour la fonction de blocage
Node.js 24	`nodejs24.x`	Amazon Linux 2	30 avril 2028	1 juin 2028	1 juillet 2028

Name	Identifiant	Système d'exploitation	Date d'obsolescence	Créer la fonction de blocage	Mettre à jour la fonction de blocage
Node.js 22	`nodejs22.x`	Amazon Linux 2	30 avril 2027	1 juin 2027	1 juillet 2027
Node.js 20	`nodejs20.x`	Amazon Linux 2	30 avril 2026	31 août 2026	30 sept. 2026
Python 3.14	`python3.14`	Amazon Linux 2	30 juin 2029	31 juillet 2029	31 août 2029
Python 3.13	`python3.13`	Amazon Linux 2	30 juin 2029	31 juillet 2029	31 août 2029
Python 3.12	`python3.12`	Amazon Linux 2	31 octobre 2028	30 novembre 2028	10 janvier 2029
Python 3.11	`python3.11`	Amazon Linux 2	30 juin 2027	31 juillet 2027	31 août 2027
Python 3.10	`python3.10`	Amazon Linux 2	31 octobre 2026	30 novembre 2026	15 janvier 2027
Java 25	`java25`	Amazon Linux 2	30 juin 2029	31 juillet 2029	31 août 2029
Java 21	`java21`	Amazon Linux 2	30 juin 2029	31 juillet 2029	31 août 2029
Java 17	`java17`	Amazon Linux 2	30 juin 2027	31 juillet 2027	31 août 2027
Java 11	`java11`	Amazon Linux 2	30 juin 2027	31 juillet 2027	31 août 2027
Java 8	`java8.al2`	Amazon Linux 2	30 juin 2027	31 juillet 2027	31 août 2027
.NET 10	`dotnet10`	Amazon Linux 2	14 novembre 2028	14 déc. 2028	15 janvier 2029

Name	Identifiant	Système d'exploitation	Date d'obsolescence	Créer la fonction de blocage	Mettre à jour la fonction de blocage
.NET 9 (conteneur uniquement)	`dotnet9`	Amazon Linux 2	10 novembre 2026	Non planifié	Non planifié
.NET 8	`dotnet8`	Amazon Linux 2	10 novembre 2026	10 déc. 2026	11 janvier 2027
Ruby 3.4	`ruby3.4`	Amazon Linux 2	31 mars 2028	30 avril 2028	31 mai 2028
Ruby 3.3	`ruby3.3`	Amazon Linux 2	31 mars 2027	30 avril 2027	31 mai 2027
Ruby 3.2	`ruby3.2`	Amazon Linux 2	31 mars 2026	31 août 2026	30 sept. 2026
Exécution réservée au système d'exploitation	`provided.al2023`	Amazon Linux 2	30 juin 2029	31 juillet 2029	31 août 2029
Exécution réservée au système d'exploitation	`provided.al2`	Amazon Linux 2	31 juillet 2026	31 août 2026	30 sept. 2026

> **Note**
>
> Pour les nouvelles régions, Lambda ne prendra pas en charge les environnements d'exécution qui devraient devenir obsolètes dans les six prochains mois.

Lambda tient à jour les environnements d'exécution gérés et de leurs images de conteneur correspondantes à jour grâce à des correctifs et à la prise en charge des versions mineures. Pour plus d'informations, consultez Mises à jour de l'exécution Lambda.

Pour interagir par programmation avec d'autres ressources Services AWS de votre fonction Lambda, vous pouvez utiliser l'un des. AWS SDKs Les environnements d'exécution Node.js, Python et Ruby incluent une version du AWS SDK. Toutefois, pour conserver le contrôle total de vos dépendances et optimiser la rétrocompatibilité lors des mises à jour automatiques de l'exécution, nous vous recommandons de toujours inclure les modules SDK utilisés par votre code (ainsi que les dépendances éventuelles) dans le package de déploiement de votre fonction ou dans une couche Lambda.

Nous vous recommandons d'utiliser la version du SDK incluant l'environnement d'exécution uniquement lorsque vous ne pouvez pas inclure de packages supplémentaires dans votre déploiement, Par exemple, lorsque vous créez votre fonction à l'aide de l'éditeur de code de la console Lambda ou à l'aide du code de fonction intégré dans un modèle. CloudFormation

Lambda met régulièrement à jour les versions AWS SDKs incluses dans les environnements d'exécution Node.js, Python et Ruby. Pour déterminer la version du SDK AWS incluse dans l'environnement d'exécution que vous utilisez, consultez les sections suivantes :

- Versions du SDK incluses dans l'environnement d'exécution (Node.js)
- Versions du SDK incluses dans l'environnement d'exécution (Python)
- Versions du SDK incluses dans l'environnement d'exécution (Ruby)

Lambda continue de prendre en charge le langage de programmation Go après l'obsolescence de l'environnement d'exécution Go 1.x. Pour plus d'informations, consultez la section Migration des AWS Lambda fonctions de l'environnement d'exécution Go1.x vers le runtime personnalisé sur Amazon Linux 2 sur le AWS blog Compute.

Tous les environnements d'exécution Lambda pris en charge supportent les architectures x86_64 et arm64.

Nouvelles versions d'exécution

Lambda fournit des environnements d'exécution gérés pour les nouvelles versions de langage uniquement lorsque la version atteint la phase LTS (support à long terme) du cycle de distribution du langage. Par exemple, pour le cycle de distribution de Node.js, lorsque la version atteint la phase Active LTS.

Avant que la version n'atteigne la phase de support à long terme, elle reste en développement et peut encore faire l'objet de modifications majeures. Lambda applique automatiquement les mises à jour

d’environnements d'exécution par défaut, de sorte que l'interruption des modifications apportées à une version d’environnement d'exécution peut empêcher vos fonctions de fonctionner comme prévu.

Lambda ne fournit pas d'environnements d'exécution gérés pour les versions linguistiques dont la sortie du LTS n'est pas prévue.

La liste suivante montre le mois de lancement cible pour les environnements d’exécution Lambda à venir. Ces dates ne sont données qu’à titre indicatif et sont susceptibles d’être modifiées.

- Ruby 3.5 - mars 2026
- Java 8, 11 et 17 du AL2 23 au deuxième trimestre 2026
- Node.js 26 - novembre 2026
- Python 3.15 - novembre 2026

politique d’obsolescence de l’exécution

Les environnements d’exécution Lambda pour les archives de fichiers .zip combinent un système d’exploitation, un langage de programmation et des bibliothèques de logiciels qui font l’objet d’une maintenance et de mises à jour de sécurité. La stratégie d’obsolescence standard de Lambda consiste à rendre obsolète un environnement d’exécution lorsqu’un composant majeur de celui-ci atteint la fin du support communautaire à long terme (LTS) et que les mises à jour de sécurité ne sont plus disponibles. Le plus souvent, il s’agit de l’environnement d’exécution du langage, bien que dans certains cas, un environnement d’exécution puisse être obsolète car le système d’exploitation (OS) atteint la fin du LTS.

Une fois qu'un environnement d'exécution est obsolète, il ne AWS peut plus y appliquer de correctifs ou de mises à jour de sécurité, et les fonctions utilisant cet environnement d'exécution ne sont plus éligibles au support technique. Ces environnements d’exécution obsolètes sont fournis « en l’état », sans aucune garantie, et peuvent contenir des bogues, des erreurs, des défauts ou d’autres vulnérabilités.

Pour en savoir plus sur la gestion des mises à niveau d'exécution et de la dépréciation, consultez les sections suivantes et la gestion des mises à niveau AWS Lambda d'exécution sur le blog AWS Compute.

> **Important**
>
> Lambda retarde parfois l'obsolescence d'un environnement d'exécution Lambda pendant une période limitée au-delà de la date de fin du support de la version du langage prise en charge par l'environnement d'exécution. Pendant cette période, Lambda applique uniquement des correctifs de sécurité au système d'exploitation de l'environnement d'exécution. Lambda n'applique pas de correctifs de sécurité aux environnements d'exécution des langages de programmation une fois leur date de fin de support atteinte.

Modèle de responsabilité partagée

Lambda est responsable de la conservation et de la publication des mises à jour de sécurité pour tous les environnements d'exécution gérés et les images de base de conteneur pris en charge. Par défaut, Lambda appliquera ces mises à jour automatiquement aux fonctions utilisant des environnements d'exécution gérés. Lorsque le paramètre de mise à jour automatique de l'environnement d'exécution par défaut a été modifié, consultez le modèle de responsabilité partagée des contrôles de gestion de l'environnement d'exécution. Pour les fonctions déployées à l'aide d'images de conteneurs, vous êtes responsable de la reconstruction de l'image de conteneur de votre fonction à partir de la dernière image de base et du redéploiement de l'image de conteneur.

Lorsqu'un environnement d'exécution devient obsolète, la responsabilité de Lambda en matière de mise à jour de l'environnement d'exécution géré et des images de base de conteneur cesse. Vous êtes responsable de la mise à niveau de vos fonctions afin d'utiliser un environnement d'exécution ou une image de base compatible.

Dans tous les cas, vous êtes responsable de l'application des mises à jour de votre code de fonction, y compris ses dépendances. Vos responsabilités dans le cadre du modèle de responsabilité partagée sont résumées dans le tableau suivant.

Phase de cycle de vie de l'environnement d'exécution	Responsabilité de Lambda	Vos responsabilités
Environnement d'exécution géré pris en charge	Fournissez des mises à jour d'environnement d'exécution régulières avec des correctifs de sécurité et d'autres mises à jour.	Mettez à jour votre code de fonction, y compris ses dépendances, pour corriger les éventuelles failles de sécurité.

Phase de cycle de vie de l'environnement d'exécution	Responsabilité de Lambda	Vos responsabilités
	Appliquer les mises à jour de l'environnement d'exécution automatiquement par défaut (consultez the section called "Modes de mise à jour de l'environnement d'exécution" pour les comportements autres que ceux par défaut).	
Image de conteneur prise en charge	Fournissez des mises à jour régulières de l'image de base du conteneur avec des correctifs de sécurité et autres mises à jour.	Mettez à jour votre code de fonction, y compris ses dépendances, pour corriger les éventuelles failles de sécurité. Reconstruisez et redéployez régulièrement votre image de conteneur à l'aide de la dernière image de base.
Environnement d'exécution géré proche de l'obsolescence	Avertissez les clients avant la dépréciation de l'exécution via la documentation Tableau de bord Health, le courrier électronique et. Trusted Advisor La responsabilité des mises à jour de l'environnement d'exécution prend fin en cas d'obsolescence.	Surveillez la documentation Lambda Tableau de bord Health, les e-mails ou les informations relatives à la Trusted Advisor dépréciation de l'exécution. Mettez à niveau les fonctions vers un environnement d'exécution compatible avant que le précédent ne soit obsolète.

Phase de cycle de vie de l'environnement d'exécution	Responsabilité de Lambda	Vos responsabilités
L'image de conteneur approche de l'obsolescence	Les notifications d'obsoles cence ne sont pas disponibles pour les fonctions utilisant des images de conteneur. La responsabilité des mises à jour de l'image de base du conteneur s'arrête à la date d'obsolescence.	Tenez compte des planifica tions d'obsolescence et mettez à niveau les fonctions vers une image de base prise en charge avant que l'image précédente ne soit obsolète.

Environnement d'exécution utilisé après la date d'obsolescence

Une fois qu'un environnement d'exécution est obsolète, il ne AWS peut plus y appliquer de correctifs ou de mises à jour de sécurité, et les fonctions utilisant cet environnement d'exécution ne sont plus éligibles au support technique. Bien que vous puissiez continuer à invoquer vos fonctions indéfiniment, il AWS est vivement recommandé de migrer vers un environnement d'exécution compatible. Les environnements d'exécution obsolètes sont fournis « en l'état », sans aucune garantie, et peuvent contenir des bogues, des erreurs, des défauts ou d'autres vulnérabilités. Les fonctions qui utilisent un environnement d'exécution obsolète peuvent subir une dégradation des performances ou d'autres problèmes, tels que l'expiration d'un certificat, qui peuvent les empêcher de fonctionner correctement.

Vous pouvez mettre à jour une fonction pour qu'elle utilise un environnement d'exécution plus récent à tout moment après qu'un environnement d'exécution soit devenu obsolète. Nous vous recommandons de tester votre fonction avec le nouveau moteur d'exécution avant d'appliquer les modifications aux environnements de production. Vous ne pourrez plus revenir à l'environnement d'exécution obsolète une fois les mises à jour des fonctions bloquées. Nous vous recommandons d'utiliser des versions et alias de fonctions pour permettre un déploiement sécurisé avec restauration.

La chronologie suivante décrit ce qui se produit lorsqu'un environnement d'exécution est obsolète :

Phase de cycle de vie de l'environnement d'exécution	Lorsque	Quoi
Période d'avis d'obsolescence	Au moins 180 jours avant l'obsolescence	• AWS envoie des notifications par e-mail et Tableau de bord Health aux comptes dont les fonctions utilisent ce runtime dans leur `$LATEST` version. • Les fonctions concernées sont également répertoriées dans l'onglet Modifications planifiées de Tableau de bord Health et dans la vérification des environnements d'exécution obsolètes AWS Trusted Advisor.
Obsolescence	Date d'obsolescence	• AWS peut ne plus appliquer les mises à jour de sécurité ou autres mises à jour. • Les fonctions ne sont plus éligibles au support technique. • Vous ne pouvez plus créer ou mettre à jour de fonctions à l'aide de l'environnement d'exécution obsolète de la console Lambda. Vous pouvez continuer à créer et à mettre à jour des fonctions via le AWS CLI AWS SAM, ou CloudFormation.
Créer la fonction de blocage	Au moins 30 jours après l'obsolescence	• Lambda commence à bloquer la création de nouvelles fonctions. • Vous pouvez continuer à mettre à jour le code et la configuration des fonctions existantes par le biais du AWS CLI AWS SAM, ou CloudFormation.
Mettre à jour la fonction de blocage	Au moins 60 jours après l'obsolescence	• Lambda commence à bloquer la mise à jour du code et de la configuration pour les fonctions existantes.

Phase de cycle de vie de l'environnement d'exécution	Lorsque	Quoi
		• Vous pouvez toujours mettre à niveau la configuration des fonctions vers un environnement d'exécution compatible. Cependant, le retour à l'environnement d'exécution obsolète pourrait être bloqué.

> **Note**
>
> Pour certains environnements d'exécution, AWS cela retarde les block-function-update dates block-function-create et au-delà des 30 et 60 jours habituels après la dépréciation. AWS a apporté cette modification en réponse aux commentaires des clients afin de vous donner plus de temps pour améliorer vos fonctionnalités. Reportez-vous aux tableaux figurant dans the section called "Environnements d'exécution pris en charge" et the section called "Environnements d'exécution obsolètes" pour connaître les dates relatives à votre environnement d'exécution. Lambda ne commencera pas à bloquer les créations ou les mises à jour de fonctions avant les dates indiquées dans ces tables.

Réception de notifications d'obsolescence lors de l'exécution

Lorsqu'un environnement d'exécution approche de sa date d'obsolescence, Lambda vous envoie une alerte par e-mail si l'une de vos Compte AWS fonctions utilise cet environnement d'exécution. Les notifications sont également affichées dans Tableau de bord Health et dans AWS Trusted Advisor.

- Réception de notifications par e-mail :

  Lambda vous envoie une alerte par e-mail au moins 180 jours avant qu'un environnement d'exécution ne soit obsolète. Cet e-mail répertorie les versions $LATEST de toutes les fonctions utilisant le runtime. Pour consulter la liste complète des versions de fonctions concernées, utilisez Trusted Advisor ou consultezthe section called "Obtenir des données sur les fonctions par environnement d'exécution".

Lambda envoie une notification par e-mail au contact principal Compte AWS de votre compte. Pour plus d'informations sur l'affichage ou la mise à jour des adresses e-mail de votre compte, consultez la section Mise à jour des informations de contact dans la Référence générale AWS .

- Recevoir des notifications par le biais de Tableau de bord Health :

  Tableau de bord Health Affiche une notification au moins 180 jours avant qu'un environnement d'exécution ne soit obsolète. Les notifications apparaissent sur la page État de votre compte sous Autres notifications. L'onglet Ressources affectées de la notification répertorie les versions $LATEST de toutes les fonctions utilisant le runtime.

  > **Note**
  >
  > Pour voir l'intégralité et la up-to-date liste des versions des fonctions concernées, utilisez Trusted Advisor ou consultezthe section called “Obtenir des données sur les fonctions par environnement d’exécution”.

  Tableau de bord Health les notifications expirent 90 jours après que le runtime concerné soit devenu obsolète.

- En utilisant AWS Trusted Advisor

  Trusted Advisor affiche une notification au moins 180 jours avant qu'un environnement d'exécution ne soit obsolète. Les notifications apparaissent sur la page Sécurité. La liste des fonctions concernées s'affiche sous Fonctions AWS Lambda utilisant des environnements d'exécution obsolètes. Cette liste de fonctions affiche à la fois $LATEST et les versions publiées et les mises à jour automatiques pour refléter l'état actuel de vos fonctions.

  Vous pouvez activer les notifications hebdomadaires par e-mail depuis Trusted Advisor la page des préférences de la Trusted Advisor console.

Environnements d’exécution obsolètes

Les environnements d’exécution suivants ont atteint la fin de prise en charge :

Name	Identifiant	Système d'exploitation	Date d'obsolescence	Créer la fonction de blocage	Mettre à jour la fonction de blocage
Python 3.9	`python3.9`	Amazon Linux 2	15 déc. 2025	31 août 2026	30 sept. 2026
Node.js 18	`nodejs18.x`	Amazon Linux 2	1e septembre 2	31 août 2026	30 sept. 2026
.NET 6	`dotnet6`	Amazon Linux 2	20 décembre 2(	31 août 2026	30 sept. 2026
Python 3.8	`python3.8`	Amazon Linux 2	14 octobre 202	31 août 2026	30 sept. 2026
Node.js 16	`nodejs16.x`	Amazon Linux 2	12 juin 2024	31 août 2026	30 sept. 2026
.NET 7 (conteneur uniquement)	`dotnet7`	Amazon Linux 2	14 mai 2024	N/A	N/A
Java 8	`java8`	Amazon Linux	8 janvier 2024	8 février 2024	30 sept. 2026
Go 1.x	`go1.x`	Amazon Linux	8 janvier 2024	8 février 2024	30 sept. 2026
Exécution réservée au système d'exploitation	`provided`	Amazon Linux	8 janvier 2024	8 février 2024	30 sept. 2026
Ruby 2.7	`ruby2.7`	Amazon Linux 2	7 décembre 20	9 janvier 2024	30 sept. 2026
Node.js 14	`nodejs14.x`	Amazon Linux 2	4 décembre 20	9 janvier 2024	30 sept. 2026

Name	Identifiant	Système d'exploitation	Date d'obsolescence	Créer la fonction de blocage	Mettre à jour la fonction de blocage
Python 3.7	`python3.7`	Amazon Linux	4 décembre 20	9 janvier 2024	30 sept. 2026
.NET Core 3.1	`dotnetcore3.1`	Amazon Linux 2	3 avril 2023	3 avril 2023	3 mai 2023
Node.js 12	`nodejs12.x`	Amazon Linux 2	31 mars 2023	31 mars 2023	30 avril 2023
Python 3.6	`python3.6`	Amazon Linux	18 juillet 2022	18 juillet 2022	29 août 2022
.NET 5 (conteneur uniquement)	`dotnet5.0`	Amazon Linux 2	10 mai 2022	N/A	N/A
.NET Core 2.1	`dotnetcore2.1`	Amazon Linux	5 janvier 2022	5 janvier 2022	13 avril 2022
Node.js 10	`nodejs10.x`	Amazon Linux 2	30 juillet 2021	30 juillet 2021	14 février 2022
Ruby 2.5	`ruby2.5`	Amazon Linux	30 juillet 2021	30 juillet 2021	31 mars 2022
Python 2.7	`python2.7`	Amazon Linux	15 juillet 2021	15 juillet 2021	30 mai 2022
Node.js 8.10	`nodejs8.10`	Amazon Linux	6 mars 2020	4 février 2020	6 mars 2020
Node.js 4.3	`nodejs4.3`	Amazon Linux	5 mars 2020	3 février 2020	5 mars 2020

Name	Identifiant	Système d'exploitation	Date d'obsolescence	Créer la fonction de blocage	Mettre à jour la fonction de blocage
Node.js 4.3 edge	`nodejs4.3-edge`	Amazon Linux	5 mars 2020	31 mars 2019	30 avril 2019
Node.js 6.10	`nodejs6.10`	Amazon Linux	12 août 2019	12 juillet 2019	12 août 2019
.NET Core 1.0	`dotnetcore1.0`	Amazon Linux	27 juin 2019	30 juin 2019	30 juillet 2019
.NET Core 2.0	`dotnetcore2.0`	Amazon Linux	30 mai 2019	30 avril 2019	30 mai 2019
Node.js 0.10	`nodejs`	Amazon Linux	30 août 2016	30 sept. 2016	31 octobre 2016

Dans presque tous les cas, la end-of-life date d'une version linguistique ou d'un système d'exploitation est connue bien à l'avance. Les liens suivants fournissent des end-of-life plannings pour chaque langage pris en charge par Lambda en tant qu'environnement d'exécution géré.

Stratégies de prise en charge des langages et des infrastructures

- Node.js – github.com
- Python – devguide.python.org
- Ruby – www.ruby-lang.org
- Java — www.oracle.com et Corretto FAQs
- Go – golang.org
- .NET – dotnet.microsoft.com

Comprendre comment Lambda gère les mises à jour de version de l'environnement d'exécution

Lambda maintient à jour chaque environnement d'exécution géré avec des mises à jour de sécurité, des corrections de bogues, de nouvelles fonctionnalités, des améliorations de performance et la prise en charge des versions mineures. Ces mises à jour d'environnement d'exécution sont publiées sous forme de versions d'environnement d'exécution. Lambda applique les mises à jour d'environnement d'exécution aux fonctions en faisant migrer la fonction d'une version antérieure vers une nouvelle version.

Par défaut, pour les fonctions utilisant des exécutions gérées, Lambda applique des mises à jour automatiquement. Avec les mises à jour de l'environnement d'exécution automatiques, Lambda prend en charge la charge opérationnelle de l'application de correctifs aux versions de l'environnement d'exécution. Pour la plupart des clients, les mises à jour automatiques sont le bon choix. Vous pouvez modifier ce comportement par défaut en configurant les paramètres de gestion de l'environnement d'exécution.

Lambda publie également chaque nouvelle version de l'environnement d'exécution sous forme d'image de conteneur. Pour mettre à jour les versions des environnements d'exécution pour les fonctions basées sur des conteneurs, vous devez créer une nouvelle image de conteneur à partir de l'image de base mise à jour et redéployer votre fonction.

Chaque version de l'environnement d'exécution est associée à un numéro de version et à un ARN (Amazon Resource Name). Les numéros de version de l'environnement d'exécution utilisent un schéma de numérotation que Lambda définit, indépendamment des numéros de version utilisés par le langage de programmation. Les numéros de version des environnements d'exécution ne sont pas toujours séquentiels. Par exemple, la version 42 peut être suivie de la version 45. L'ARN de la version de l'environnement d'exécution est un identifiant unique pour chaque version. Vous pouvez voir l'ARN de la version de l'environnement d'exécution actuelle de votre fonction dans la console Lambda ou dans la ligne `INIT_START` des journaux de votre fonction.

Les versions des environnements d'exécution ne doivent pas être confondues avec les identifiants des environnements d'exécution. Chaque environnement d'exécution possède un identifiant unique, tel que `python3.14` ou `nodejs24.x`. Ceux-ci correspondent à chaque version majeure du langage de programmation. Les versions de l'environnement d'exécution décrivent la version corrective d'un environnement d'exécution individuel.

> **Note**
>
> L'ARN pour le même numéro de version d'exécution peut varier selon Régions AWS les architectures de processeur.

Rubriques

- Rétrocompatibilité
- Modes de mise à jour de l’environnement d’exécution
- Déploiement des versions de l’environnement d’exécution en deux phases
- Configuration des paramètres de gestion de l’environnement d’exécution Lambda
- Rétablissement d’une version de l’environnement d’exécution Lambda
- Identification des changements de version de l’environnement d’exécution Lambda
- Comprendre le modèle de responsabilité partagée pour la gestion des environnements d’exécution Lambda
- Contrôle des autorisations de mise à jour de l’environnement d’exécution Lambda pour les applications hautement conformes

Rétrocompatibilité

Lambda s’efforce de fournir des mises à jour d’environnement d’exécution qui sont rétrocompatibles avec les fonctions existantes. Cependant, comme pour les correctifs logiciels, il existe de rares cas dans lesquels une mise à jour de l’environnement d’exécution peut avoir un impact négatif sur une fonction existante. Par exemple, les correctifs de sécurité peuvent exposer un problème sous-jacent à une fonction existante qui dépend du comportement précédent, non sécurisé.

Lorsque vous créez et déployez votre fonction, il est important de comprendre comment gérer vos dépendances afin d’éviter d’éventuelles incompatibilités lors d’une future mise à jour d’environnement d’exécution. Supposons, par exemple, que votre fonction soit dépendante du package A, lui-même dépendant du package B. Les deux packages sont inclus dans l’environnement d’exécution Lambda (par exemple, ils peuvent faire partie du SDK ou de ses dépendances, ou faire partie des bibliothèques du système d’exécution).

Réfléchissez aux scénarios suivants :

Déploiement	Compatible avec les correctifs	Raison
• Package A : utilisation depuis l'environnement d'exécution • Package B : utilisation depuis l'environnement d'exécution	Oui	Les futures mises à jour d'environnement d'exécution des packages A et B seront rétrocompatibles.
• Package A : dans le package de déploiement • Package B : dans le package de déploiement	Oui	Votre déploiement étant prioritaire, les futures mises à jour d'environnement d'exécuti on des packages A et B n'auront aucun effet.
• Package A : dans le package de déploiement • Package B : utilisation depuis l'environnement d'exécution	Oui*	Les futures mises à jour d'environnement d'exécuti on du package B seront rétrocompatibles. *Si A et B sont étroitement couplés, des problèmes de compatibilité peuvent survenir. Par exemple, les `botocore` packages `boto3` et du AWS SDK pour Python doivent être déployés ensemble.
• Package A : utilisation depuis l'environnement d'exécution • Package B : dans le package de déploiement	Non	Les futures mises à jour d'environnement d'exécuti on du package A peuvent nécessiter une version mise à jour du package B. Cependant , la version déployée du package B est prioritaire et risque de ne pas être

Déploiement	Compatible avec les correctifs	Raison
		compatible avec la version mise à jour du package A.

Pour maintenir la compatibilité avec les futures mises à jour d'environnement d'exécution, suivez les bonnes pratiques suivantes :

- Dans la mesure du possible, regroupez toutes les dépendances : incluez toutes les bibliothèques requises, y compris le AWS SDK et ses dépendances, dans votre package de déploiement. Cela garantit que votre ensemble de composants reste stable et compatible.
- Utilisez l'environnement d'exécution SDKs avec parcimonie : utilisez uniquement le SDK fourni par l'exécution lorsque vous ne pouvez pas inclure de packages supplémentaires (par exemple, lorsque vous utilisez l'éditeur de code de la console Lambda ou du code en ligne dans un modèle). AWS CloudFormation
- Évitez de remplacer les bibliothèques système : ne déployez pas de bibliothèques de système d'exploitation personnalisées susceptibles d'entrer en conflit avec les futures mises à jour d'environnement d'exécution.

Modes de mise à jour de l'environnement d'exécution

Lambda s'efforce de fournir des mises à jour d'environnement d'exécution qui sont rétrocompatibles avec les fonctions existantes. Cependant, comme pour les correctifs logiciels, il existe de rares cas dans lesquels une mise à jour de l'environnement d'exécution peut avoir un impact négatif sur une fonction existante. Par exemple, les correctifs de sécurité peuvent exposer un problème sous-jacent à une fonction existante qui dépend du comportement précédent, non sécurisé. Les contrôles de gestion de l'environnement d'exécution Lambda permettent de réduire le risque d'impact sur vos charges de travail dans le cas rare d'une incompatibilité de version. Pour chaque version de fonction ($LATEST ou version publiée), vous pouvez choisir l'un des modes de mise à jour de l'environnement d'exécution suivants :

- Auto (par défaut) – Mettre automatiquement à jour vers la version de l'environnement d'exécution la plus sécurisée et la plus sûre à l'aide de Déploiement des versions de l'environnement d'exécution en deux phases. Nous recommandons ce mode à la plupart des clients afin que vous puissiez toujours bénéficier des mises à jour de l'environnement d'exécution.

- Mise à jour de fonction – Mise à jour vers la version d'exécution la plus récente et la plus sécurisée lorsque vous mettez à jour votre fonction. Lorsque vous mettez à jour votre fonction, Lambda met à jour l'exécution de votre fonction vers la version la plus récente et la plus sécurisée. Cette approche synchronise les mises à jour de l'environnement d'exécution avec les déploiements de fonctions, ce qui vous permet de contrôler le moment où Lambda applique les mises à jour de l'environnement d'exécution. Avec ce mode, vous pouvez détecter et atténuer rapidement les rares incompatibilités de mise à jour de l'environnement d'exécution. Lorsque vous utilisez ce mode, vous devez régulièrement mettre à jour vos fonctions pour maintenir leur environnement d'exécution à jour.
- Manuel – Mise à jour manuelle de la version de votre exécution. Vous spécifiez une version de l'exécution dans la configuration de votre fonction. La fonction utilise cette version de l'environnement d'exécution indéfiniment. Dans les rares cas où une nouvelle version de l'environnement d'exécution est incompatible avec une fonction existante, vous pouvez utiliser ce mode pour ramener votre fonction à une version antérieure. Nous vous déconseillons d'utiliser le mode Manual (Manuel) pour tenter d'obtenir une cohérence de l'environnement d'exécution entre les déploiements. Pour de plus amples informations, veuillez consulter Rétablissement d'une version de l'environnement d'exécution Lambda.

La responsabilité de l'application des mises à jour de l'environnement d'exécution à vos fonctions varie en fonction du mode de mise à jour que vous choisissez. Pour de plus amples informations, veuillez consulter Comprendre le modèle de responsabilité partagée pour la gestion des environnements d'exécution Lambda.

Déploiement des versions de l'environnement d'exécution en deux phases

Lambda introduit de nouvelles gestions des versions d'exécution dans l'ordre suivant :

1. Dans la première phase, Lambda applique la nouvelle version d'exécution chaque fois que vous créez ou mettez à jour une fonction. Une fonction est mise à jour lorsque vous appelez les opérations de UpdateFunctionConfigurationl'API UpdateFunctionCodeor.
2. Dans la deuxième phase, Lambda met à jour toute fonction qui utilise le mode de mise à jour Auto de l'environnement d'exécution et qui n'a pas encore été mise à jour vers la nouvelle version.

La durée totale du processus de déploiement varie en fonction de plusieurs facteurs, notamment la gravité de tout correctif de sécurité inclus dans la mise à jour de l'environnement d'exécution.

Si vous développez et déployez activement vos fonctions, vous obtiendrez très probablement de nouvelles versions de l'environnement d'exécution au cours de la première phase. Cela permet de synchroniser les mises à jour de l'environnement d'exécution avec les mises à jour des fonctions. Dans le cas rare où la dernière version d'exécution aurait un impact négatif sur votre application, cette approche vous permet de prendre rapidement des mesures correctives. Les fonctions qui ne sont pas en cours de développement bénéficient toujours des avantages opérationnels des mises à jour automatiques de l'environnement d'exécution au cours de la deuxième phase.

Cette approche n'affecte pas les fonctions définies en mode Function update (Mise à jour de fonction) ou en mode Manual (Manuel). Les fonctions utilisant le mode Function update (Mise à jour de fonction) reçoivent les dernières mises à jour de l'environnement d'exécution uniquement lorsque vous les créez ou les mettez à jour. Les fonctions utilisant le mode Manual (Manuel) ne reçoivent pas de mises à jour de l'environnement d'exécution.

Lambda publie les nouvelles versions de l'environnement d'exécution de manière progressive et continue à travers Régions AWS. Si vos fonctions sont configurées en mode Auto ou Function update (Mise à jour de fonction), il est possible que les fonctions déployées au même moment dans différentes régions, ou à différents moments dans la même région, reçoivent des versions de l'environnement d'exécution différentes. Les clients qui exigent une cohérence garantie des versions de l'environnement d'exécution dans leurs environnements doivent utiliser des images de conteneurs pour déployer leurs fonctions Lambda. Le mode Manuel est conçu comme une mesure d'atténuation temporaire pour permettre de restaurer la version d'exécution dans le rare cas où une version d'exécution est incompatible avec votre fonction.

Configuration des paramètres de gestion de l'environnement d'exécution Lambda

Vous pouvez configurer les paramètres de gestion de l'environnement d'exécution à l'aide de la console Lambda ou de AWS Command Line Interface (AWS CLI).

> **Note**
>
> Vous pouvez configurer les paramètres de gestion de l'environnement d'exécution séparément pour chaque version de fonction.

Pour configurer la façon dont Lambda met à jour la version de votre environnement d'exécution (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom d'une fonction.
3. Dans l'onglet Code, sous Runtime settings (Paramètres de l'environnement d'exécution), choisissez Edit runtime management configuration (Modifier la configuration de gestion de l'environnement d'exécution).
4. Sous Runtime management configuration (Configuration de la gestion de l'environnement d'exécution), choisissez l'une des options suivantes :

   - Pour que votre fonction se mette automatiquement à jour vers la dernière version de l'environnement d'exécution, sélectionnez Auto.
   - Pour que votre fonction se mette à jour vers la dernière version de l'environnement d'exécution lorsque vous modifiez la fonction, sélectionnez Function update (Mise à jour de fonction).
   - Pour que votre fonction soit mise à jour vers la dernière version de l'environnement d'exécution uniquement lorsque vous modifiez la version de l'environnement d'exécution ARN, sélectionnez Manual (Manuel). Vous trouverez l'ARN de la version de l'environnement d'exécution sous Runtime management configuration (Configuration de la gestion de l'environnement d'exécution). Vous pouvez également trouver l'ARN dans la ligne `INIT_START` des journaux de votre fonction.

   Pour de plus amples informations sur ces options, veuillez consulter Modes de mise à jour de l'environnement d'exécution.
5. Choisissez Enregistrer.

Pour configurer la façon dont Lambda met à jour votre version de l'environnement d'exécution (AWS CLI)

Pour configurer la gestion de l'environnement d'exécution d'une fonction, exécutez la commande put-runtime-management-config de l'AWS CLI. Lorsque vous utilisez le mode `Manual`, vous devez également fournir l'ARN de la version de l'environnement d'exécution.

```
aws lambda put-runtime-management-config \
  --function-name my-function \
  --update-runtime-on Manual \
```

```
  --runtime-version-arn arn:aws:lambda:us-
east-2::runtime:8eeff65f6809a3ce81507fe733fe09b835899b99481ba22fd75b5a7338290ec1
```

Vous devez voir des résultats similaires à ce qui suit :

```
{
  "UpdateRuntimeOn": "Manual",
  "FunctionArn": "arn:aws:lambda:us-east-2:111122223333:function:my-function",
  "RuntimeVersionArn": "arn:aws:lambda:us-
east-2::runtime:8eeff65f6809a3ce81507fe733fe09b835899b99481ba22fd75b5a7338290ec1"
}
```

Rétablissement d'une version de l'environnement d'exécution Lambda

Dans le cas rare où une nouvelle version de l'environnement d'exécution est incompatible avec votre fonction existante, vous pouvez rétablir sa version à une version antérieure. Cela permet à votre application de continuer à fonctionner et de minimiser les perturbations, en laissant le temps de remédier à l'incompatibilité avant de revenir à la dernière version de l'environnement d'exécution.

Lambda n'impose pas de limite de temps à l'utilisation d'une version de l'environnement d'exécution particulière. Cependant, nous vous recommandons vivement de passer à la dernière version de l'environnement d'exécution dès que possible pour bénéficier des derniers correctifs de sécurité, des améliorations de performances et des fonctionnalités. Lambda offre la possibilité de revenir à une version antérieure de l'environnement d'exécution uniquement à titre d'atténuation temporaire dans le cas rare d'un problème de compatibilité de mise à jour de l'environnement d'exécution. Les fonctions qui utilisent une version d'exécution antérieure pendant une période prolongée peuvent subir une dégradation des performances ou des problèmes, tels que l'expiration d'un certificat, qui peuvent les empêcher de fonctionner correctement.

Vous pouvez rétablir une version de l'environnement d'exécution de l'une des manières suivantes :

- En utilisant le mode de mise à jour Manual (Manuel) de l'environnement d'exécution
- En utilisant les versions de la fonction publiée

Pour plus d'informations, consultez Introduction aux contrôles de gestion de l'environnement d'exécution AWS Lambda sur le blog AWS.

Rétablissement d'une version de l'environnement d'exécution à l'aide du mode de mise à jour Manual (Manuel)

Si vous utilisez le mode de mise à jour Auto de la version de l'environnement d'exécution, ou si vous utilisez la version `$LATEST`, vous pouvez rétablir la version de votre environnement d'exécution en utilisant le mode Manual (Manuel). Pour la version de la fonction que vous voulez rétablir, définissez le mode de mise à jour de la version de l'environnement d'exécution sur Manual (Manuel) et indiquez l'ARN de la version précédente. Pour plus d'informations sur la recherche de l'ARN de la version de l'environnement d'exécution précédente, consultez Identification des changements de version de l'environnement d'exécution Lambda.

> **Note**
>
> Si la version `$LATEST` de votre fonction est configurée pour utiliser le mode Manual (Manuel), vous ne pouvez pas modifier l'architecture du processeur ou la version de l'environnement d'exécution que votre fonction utilise. Pour effectuer ces modifications, vous devez passer en mode Auto ou Function update (Mise à jour de fonction).

Rétablissement d'une version de l'environnement d'exécution à l'aide des versions de fonction publiées

Les versions de fonction publiées sont un instantané immuable du code et de la configuration de la fonction `$LATEST` au moment où vous les avez créées. En mode Auto, Lambda met automatiquement à jour la version de l'environnement d'exécution des versions de fonctions publiées pendant la phase deux du déploiement de la version de l'environnement d'exécution. En mode Function update (Mise à jour de fonction), Lambda ne met pas à jour la version de l'environnement d'exécution des versions de fonctions publiées.

Les versions de fonctions publiées utilisant le mode Function update (Mise à jour de fonction) créent donc un instantané statique du code de la fonction, de la configuration et de la version de l'environnement d'exécution. En utilisant le mode Function update (Mise à jour de fonction) avec les versions de fonctions, vous pouvez synchroniser les mises à jour de l'environnement d'exécution avec vos déploiements. Vous pouvez également coordonner le rétablissement des versions de code, de configuration et de l'environnement d'exécution en redirigeant le trafic vers une version de fonction publiée antérieurement. Vous pouvez intégrer cette approche à votre système d'intégration et de livraison continues (CI/CD) afin de rétablir automatiquement la situation dans

les rares cas d'incompatibilité des mises à jour de l'environnement d'exécution. Lorsque vous utilisez cette approche, vous devez mettre à jour votre fonction régulièrement et publier de nouvelles versions de la fonction pour récupérer les dernières mises à jour de l'exécution. Pour de plus amples informations, consultez Comprendre le modèle de responsabilité partagée pour la gestion des environnements d'exécution Lambda.

Identification des changements de version de l'environnement d'exécution Lambda

Le numéro de version de l'environnement d'exécution et l'ARN sont consignés dans la ligne de journal `INIT_START`, que Lambda émet vers CloudWatch Logs chaque fois qu'il crée un nouvel environnement d'exécution. Étant donné que l'environnement d'exécution utilise la même exécution pour tous les invocations de fonction, Lambda émet la ligne de journal `INIT_START` uniquement lorsqu'il exécute la phase init. Lambda n'émet pas cette ligne de journal pour chaque invocation de fonction. Lambda émet la ligne de journal vers CloudWatch Logs, mais elle n'est pas visible dans la console.

> **Note**
>
> Les numéros de version des environnements d'exécution ne sont pas toujours séquentiels. Par exemple, la version 42 peut être suivie de la version 45.

Example Exemple de ligne de journal INIT_START

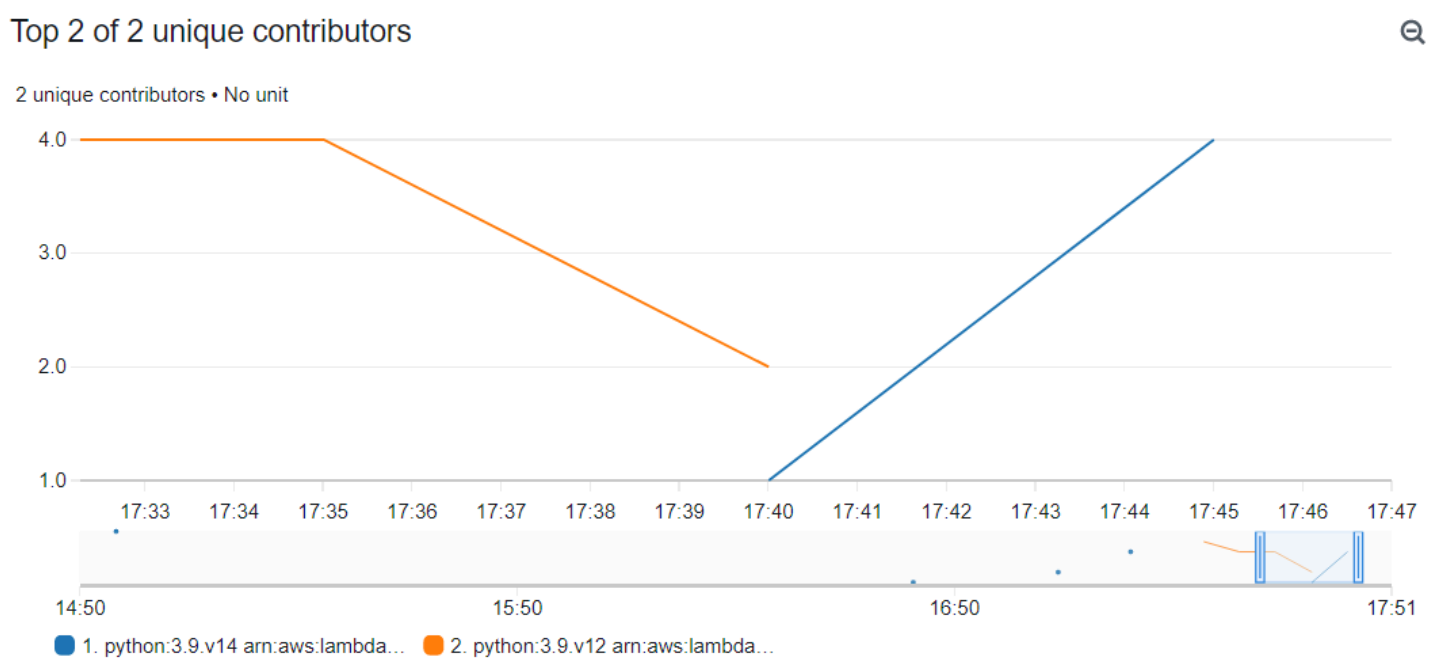
```
INIT_START Runtime Version: python:3.13.v14    Runtime Version ARN: arn:aws:lambda:eu-
south-1::runtime:7b620fc2e66107a1046b140b9d320295811af3ad5d4c6a011fad1fa65127e9e6I
```

Plutôt que de travailler directement avec les journaux, vous pouvez utiliser Amazon CloudWatch Contributor Insights pour identifier les transitions entre les versions de l'environnement d'exécution. La règle suivante compte les versions de l'environnement d'exécution distinctes de chaque ligne de journal `INIT_START`. Pour utiliser la règle, remplacez l'exemple de nom de groupe de journaux `/aws/lambda/*` par le préfixe approprié pour votre fonction ou groupe de fonctions.

```
{
  "Schema": {
    "Name": "CloudWatchLogRule",
    "Version": 1
```

```
  },
  "AggregateOn": "Count",
  "Contribution": {
    "Filters": [
      {
        "Match": "eventType",
        "In": [
          "INIT_START"
        ]
      }
    ],
    "Keys": [
      "runtimeVersion",
      "runtimeVersionArn"
    ]
  },
  "LogFormat": "CLF",
  "LogGroupNames": [
    "/aws/lambda/*"
  ],
  "Fields": {
    "1": "eventType",
    "4": "runtimeVersion",
    "8": "runtimeVersionArn"
  }
}
```

Le rapport suivant de CloudWatch Contributor Insights montre un exemple de transition de version de l'environnement d'exécution tel que capturé par la règle précédente. La ligne orange montre l'initialisation de l'environnement d'exécution pour la version précédente (python:3.13.v12), et la ligne bleue montre l'initialisation de l'environnement d'exécution pour la nouvelle version (python:3.13.v14).



Comprendre le modèle de responsabilité partagée pour la gestion des environnements d'exécution Lambda

Lambda est responsable de la conservation et de la publication des mises à jour de sécurité pour tous les environnements d'exécution gérés et les images de conteneurs pris en charge. La responsabilité de la mise à jour des fonctions existantes pour utiliser la dernière version de l'environnement d'exécution varie en fonction du mode de mise à jour que vous utilisez.

Lambda est responsable de l'application des mises à jour de l'environnement d'exécution à toutes les fonctions configurées pour utiliser le mode de mise à jour Auto.

Pour les fonctions configurées avec le mode de mise à jour d'exécution Function update (Mise à jour de fonction), vous êtes responsable de la mise à jour régulière de votre fonction. Lambda est responsable de l'application des mises à jour de l'environnement d'exécution lorsque vous effectuez ces mises à jour. Si vous ne mettez pas à jour votre fonction, Lambda ne met pas à jour l'environnement d'exécution. Si vous ne mettez pas régulièrement à jour votre fonction, nous vous recommandons vivement de la configurer pour les mises à jour automatiques de l'environnement d'exécution afin qu'elle continue à recevoir les mises à jour de sécurité.

Pour les fonctions configurées pour utiliser le mode de mise à jour Manual (Manuel), vous êtes responsable de la mise à jour de votre fonction pour utiliser la dernière version de l'environnement d'exécution. Nous vous recommandons vivement d'utiliser ce mode uniquement pour rétablir la version de l'environnement d'exécution à titre de mesure d'atténuation temporaire dans le cas rare

d'une incompatibilité de mise à jour. Nous vous recommandons également de passer en mode Auto le plus rapidement possible afin de minimiser le temps pendant lequel vos fonctions ne sont pas corrigées.

Si vous utilisez des images de conteneur pour déployer vos fonctions, Lambda est responsable de la publication des images de base mises à jour. Dans ce cas, vous êtes responsable de la recréation de l'image de conteneur de votre fonction à partir de la dernière image de base et du redéploiement de l'image de conteneur.

Ceci est résumé dans le tableau suivant :

Mode de déploiement	Responsabilité de Lambda	Responsabilité du client
Environnement d'exécution géré, mode Auto	Publiez de nouvelles versions de l'environnement d'exécution contenant les derniers correctifs. Appliquez les correctifs de l'environnement d'exécution aux fonctions existantes.	Rétablissez une version de l'environnement d'exécution précédente dans le cas rare d'un problème de compatibilité de mise à jour de l'environnement d'exécution. Suivez les meilleures pratiques de rétrocompatibilité.
Environnement d'exécution géré, mode Function update (Mise à jour de fonction)	Publiez de nouvelles versions de l'environnement d'exécution contenant les derniers correctifs.	Mettez régulièrement à jour les fonctions afin de bénéficier de la dernière version de l'environnement d'exécution. Passez une fonction en mode Auto lorsque vous ne la mettez pas régulièrement à jour. Rétablissez une version de l'environnement d'exécution précédente dans le cas rare d'un problème de compatibilité de mise à jour de l'environnement d'exécution. Suivez les meilleures pratiques de rétrocompatibilité.
Environnement d'exécuti	Publiez de nouvelles versions de l'environnement d'exécuti	Utilisez ce mode uniquement pour restaurer temporairement l'environnement d'exécution

Mode de déploiement	Responsabilité de Lambda	Responsabilité du client
on géré, mode Manual (Manuel)	on contenant les derniers correctifs.	dans le cas rare d'un problème de compatibilité de mise à jour. Passez les fonctions en mode Auto ou Function update (Mise à jour de fonction) et la dernière version de l'environnement d'exécution dès que possible.
Image de conteneur	Publiez de nouvelles images de conteneur contenant les derniers correctifs.	Redéployez régulièrement les fonctions en utilisant la dernière image de base du conteneur pour bénéficier des derniers correctif s.

Pour plus d'informations sur la responsabilité partagée avec AWS, consultez Modèle de responsabilité partagée.

Contrôle des autorisations de mise à jour de l'environnement d'exécution Lambda pour les applications hautement conformes

Pour répondre aux exigences de correctifs, les clients Lambda s'appuient généralement sur des mises à jour automatiques de l'environnement d'exécution. Si votre application est soumise à des exigences strictes en matière de nouveauté des correctifs, vous voudrez peut-être limiter l'utilisation de versions de l'environnement d'exécution antérieures. Vous pouvez restreindre les contrôles de gestion de l'environnement d'exécution de Lambda en utilisant Gestion des identités et des accès AWS (IAM) pour refuser aux utilisateurs de votre compte AWS l'accès à l'opération d'API PutRuntimeManagementConfig. Cette opération est utilisée pour choisir le mode de mise à jour de l'environnement d'exécution d'une fonction. En refusant l'accès à cette opération, toutes les fonctions passent par défaut au mode Auto. Vous pouvez appliquer cette restriction à l'échelle de votre organisation en utilisant des politiques de contrôle des services (SCP). Si vous devez rétablir une fonction à une version de l'environnement d'exécution antérieure, vous pouvez accorder une exception de politique au cas par cas.

Récupération de données sur les fonctions Lambda qui utilisent un environnement d’exécution obsolète

Lorsqu'un environnement d'exécution Lambda est sur le point de devenir obsolète, Lambda vous avertit par e-mail et vous envoie des notifications dans le et. Tableau de bord Health Trusted Advisor Ces e-mails et notifications répertorient les versions $LATEST des fonctions utilisant l’environnement d’exécution. Pour répertorier toutes les versions de vos fonctions qui utilisent un environnement d'exécution particulier, vous pouvez utiliser le AWS Command Line Interface (AWS CLI) ou l'un des AWS SDKs.

Si un grand nombre de fonctions utilisent un environnement d'exécution destiné à être obsolète, vous pouvez également utiliser le AWS CLI ou pour vous aider AWS SDKs à prioriser les mises à jour de vos fonctions les plus fréquemment invoquées.

Reportez-vous aux sections suivantes pour savoir comment utiliser AWS CLI et AWS SDKs pour collecter des données sur les fonctions qui utilisent un environnement d'exécution particulier.

Liste des versions de fonctions qui utilisent un environnement d’exécution particulier

Pour utiliser le AWS CLI pour répertorier toutes les versions de vos fonctions qui utilisent un environnement d'exécution particulier, exécutez la commande suivante. `RUNTIME_IDENTIFIER`Remplacez-le par le nom du moteur d'exécution qui est obsolète et choisissez le vôtre. Région AWS Pour répertorier uniquement les versions de fonction $LATEST, omettez `--function-version ALL` dans la commande.

```
aws lambda list-functions --function-version ALL --region us-east-1 --output text --
query "Functions[?Runtime=='RUNTIME_IDENTIFIER'].FunctionArn"
```

> **Tip**
>
> L'exemple de commande répertorie les fonctions d'une `us-east-1` région en particulier. Compte AWS Vous devrez répéter cette commande pour chaque région dans laquelle votre compte possède des fonctions et pour chacune de vos Comptes AWS.

Vous pouvez également répertorier les fonctions qui utilisent un environnement d'exécution particulier à l'aide de l'un des AWS SDKs. L'exemple de code suivant utilise le V3 AWS SDK pour

JavaScript et le AWS SDK pour Python (Boto3) pour renvoyer une liste des fonctions ARNs utilisant un environnement d'exécution particulier. L'exemple de code renvoie également le groupe de CloudWatch journaux pour chacune des fonctions répertoriées. Vous pouvez utiliser ce groupe de journaux pour trouver la date de la dernière invocation de la fonction. Consultez la section the section called “Identification des fonctions les plus fréquemment et les plus récemment invoquées” suivante pour plus d’informations.

Node.js

Example JavaScript code pour répertorier les fonctions utilisant un environnement d'exécution particulier

```
import { LambdaClient, ListFunctionsCommand } from "@aws-sdk/client-lambda";
const lambdaClient = new LambdaClient();

const command = new ListFunctionsCommand({
    FunctionVersion: "ALL",
    MaxItems: 50
});
const response = await lambdaClient.send(command);

for (const f of response.Functions){
    if (f.Runtime == '<your_runtime>'){ // Use the runtime id, e.g. 'nodejs24.x' or
 'python3.14'
        console.log(f.FunctionArn);
        // get the CloudWatch log group of the function to
        // use later for finding the last invocation date
        console.log(f.LoggingConfig.LogGroup);
    }
}
// If your account has more functions than the specified
// MaxItems, use the returned pagination token in the
// next request with the 'Marker' parameter
if ('NextMarker' in response){
    let paginationToken = response.NextMarker;
  }
```

Python

Example Code Python pour répertorier les fonctions utilisant un environnement d'exécution particulier

```
import boto3
from botocore.exceptions import ClientError

def list_lambda_functions(target_runtime):

    lambda_client = boto3.client('lambda')

    response = lambda_client.list_functions(
        FunctionVersion='ALL',
        MaxItems=50
    )
    if not response['Functions']:
            print("No Lambda functions found")
    else:
        for function in response['Functions']:
            if function['PackageType']=='Zip' and function['Runtime'] ==
 target_runtime:
                print(function['FunctionArn'])
                # Print the CloudWatch log group of the function
                # to use later for finding last invocation date
                print(function['LoggingConfig']['LogGroup'])

    if 'NextMarker' in response:
       pagination_token = response['NextMarker']

if __name__ == "__main__":
    # Replace python3.12 with the appropriate runtime ID for your Lambda functions
    list_lambda_functions('python3.12')
```

Pour en savoir plus sur l'utilisation d'un AWS SDK pour répertorier vos fonctions à l'aide de l'ListFunctionsaction, consultez la documentation du SDK correspondant à votre langage de programmation préféré.

Vous pouvez également utiliser la fonctionnalité de requêtes AWS Config avancées pour répertorier toutes les fonctions qui utilisent un environnement d'exécution affecté. Cette requête renvoie uniquement les versions $LATEST de la fonction, mais vous pouvez agréger les requêtes pour

répertorier les fonctions dans toutes les régions et plusieurs Comptes AWS avec une seule commande. Pour en savoir plus, consultez la section Interrogation de l'état de configuration actuel des AWS Auto Scaling ressources dans le guide du AWS Config développeur.

Identification des fonctions les plus fréquemment et les plus récemment invoquées

Si vous utilisez Compte AWS des fonctions qui utilisent un environnement d'exécution destiné à être obsolète, vous souhaiterez peut-être donner la priorité à la mise à jour des fonctions fréquemment invoquées ou des fonctions qui ont été invoquées récemment.

Si vous ne disposez que de quelques fonctions, vous pouvez utiliser la console CloudWatch Logs pour recueillir ces informations en consultant les flux de journaux de vos fonctions. Voir Afficher les données de journal envoyées à CloudWatch Logs pour plus d'informations.

Pour connaître le nombre d'appels de fonctions récents, vous pouvez également utiliser les informations de CloudWatch métriques affichées dans la console Lambda. Pour afficher ces informations, procédez comme suit :

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction dont vous souhaitez consulter les statistiques d'invocation.
3. Choisissez l'onglet Surveiller.
4. Définissez la période pendant laquelle vous souhaitez consulter les statistiques à l'aide du sélecteur de plage de dates. Les invocations récentes sont affichées dans le volet Invocations.

Pour les comptes dotés d'un plus grand nombre de fonctions, il peut être plus efficace de collecter ces données par programmation en utilisant l'une AWS CLI ou l'une des AWS SDKs actions d'GetMetricStatisticsAPI DescribeLogStreamset.

Les exemples suivants fournissent des extraits de code utilisant la V3 AWS SDK pour JavaScript et le AWS SDK pour Python (Boto3) pour identifier la date du dernier appel pour une fonction particulière et pour déterminer le nombre d'appels pour une fonction particulière au cours des 14 derniers jours.

Node.js

Example JavaScript code pour trouver l'heure du dernier appel d'une fonction

```
import { CloudWatchLogsClient, DescribeLogStreamsCommand } from "@aws-sdk/client-
cloudwatch-logs";
```

```
const cloudWatchLogsClient = new CloudWatchLogsClient();
const command = new DescribeLogStreamsCommand({
    logGroupName: '<your_log_group_name>',
    orderBy: 'LastEventTime',
    descending: true,
    limit: 1
});
try {
    const response = await cloudWatchLogsClient.send(command);
    const lastEventTimestamp = response.logStreams.length > 0 ?
        response.logStreams[0].lastEventTimestamp : null;
    // Convert the UNIX timestamp to a human-readable format for display
    const date = new Date(lastEventTimestamp).toLocaleDateString();
    const time = new Date(lastEventTimestamp).toLocaleTimeString();
    console.log(`${date} ${time}`);

} catch (e){
    console.error('Log group not found.')
}
```

Python

Example Code Python pour trouver l'heure de la dernière invocation d'une fonction

```
import boto3
from datetime import datetime

cloudwatch_logs_client  = boto3.client('logs')

response = cloudwatch_logs_client.describe_log_streams(
    logGroupName='<your_log_group_name>',
    orderBy='LastEventTime',
    descending=True,
    limit=1
)

try:
    if len(response['logStreams']) > 0:
        last_event_timestamp = response['logStreams'][0]['lastEventTimestamp']
        print(datetime.fromtimestamp(last_event_timestamp/1000)) # Convert timestamp
 from ms to seconds
    else:
        last_event_timestamp = None
except:
```

```
    print('Log group not found')
```

> **Tip**
>
> Vous pouvez trouver le nom du groupe de journaux de votre fonction à l'aide de l'opération ListFunctionsAPI. Consultez le code dans the section called “Liste des versions de fonctions qui utilisent un environnement d’exécution particulier” pour obtenir un exemple montrant la façon de procéder.

Node.js

Example JavaScript code pour trouver le nombre d'invocations au cours des 14 derniers jours

```
import { CloudWatchClient, GetMetricStatisticsCommand } from "@aws-sdk/client-
cloudwatch";
const cloudWatchClient = new CloudWatchClient();
const command = new GetMetricStatisticsCommand({
    Namespace: 'AWS/Lambda',
    MetricName: 'Invocations',
    StartTime: new Date(Date.now()-86400*1000*14), // 14 days ago
    EndTime: new Date(Date.now()),
    Period: 86400 * 14, // 14 days.
    Statistics: ['Sum'],
    Dimensions: [{
        Name: 'FunctionName',
        Value: '<your_function_name>'
    }]
});
const response = await cloudWatchClient.send(command);
const invokesInLast14Days = response.Datapoints.length > 0 ?
    response.Datapoints[0].Sum : 0;

console.log('Number of invocations: ' + invokesInLast14Days);
```

Python

Example Code Python pour trouver le nombre d’invocations au cours des 14 derniers jours

```
import boto3
```

```
from datetime import datetime, timedelta

cloudwatch_client = boto3.client('cloudwatch')

response = cloudwatch_client.get_metric_statistics(
    Namespace='AWS/Lambda',
    MetricName='Invocations',
    Dimensions=[
        {
            'Name': 'FunctionName',
            'Value': '<your_function_name>'
        },
    ],
    StartTime=datetime.now() - timedelta(days=14),
    EndTime=datetime.now(),
    Period=86400 * 14, # 14 days
    Statistics=[
        'Sum'
    ]
)

if len(response['Datapoints']) > 0:
    invokes_in_last_14_days = int(response['Datapoints'][0]['Sum'])
else:
    invokes_in_last_14_days = 0

print(f'Number of invocations: {invokes_in_last_14_days}')
```

Modification de l'environnement d'exécution

Vous pouvez utiliser des extensions internes pour modifier le processus de runtime. Les extensions internes ne sont pas des processus séparés. Elles s'exécutent dans le cadre du processus du runtime.

Lambda fournit des variables d'environnement spécifiques du langage, que vous pouvez définir pour ajouter des options et des outils au runtime. Lambda intègre également des scripts encapsuleurs qui lui permettent de déléguer le démarrage du runtime à votre script. Vous pouvez créer un script encapsuleur pour personnaliser le comportement de démarrage du runtime.

Variables d'environnement spécifiques à un langage

Lambda prend en charge des méthodes de configuration pour permettre le pré-chargement de code lors de l'initialisation de la fonction via les variables d'environnement spécifiques du langage suivantes :

- `JAVA_TOOL_OPTIONS` : sur Java, Lambda prend en charge cette variable d'environnement pour définir des variables de ligne de commande supplémentaires dans Lambda. Cette variable d'environnement vous permet de spécifier l'initialisation des outils, en particulier le lancement d'agents de langage de programmation natifs ou Java à l'aide des options `agentlib` ou `javaagent`. Pour plus d'informations, veuillez consulter la rubrique Variables d'environnement `JAVA_TOOL_OPTIONS`.
- `NODE_OPTIONS`— Disponible dans les environnements d'exécution de Node.js.
- `DOTNET_STARTUP_HOOKS` – Sur .NET Core 3.1 et versions ultérieures, cette variable d'environnement précise le chemin d'accès d'un assembly (dll) que Lambda peut utiliser.

L'utilisation de variables d'environnement spécifiques à un langage est la méthode privilégiée pour définir les propriétés de démarrage.

Scripts encapsuleurs

Vous pouvez créer un script encapsuleur pour personnaliser le comportement de démarrage du runtime de votre fonction Lambda. Un script encapsuleur vous permet de définir des paramètres de configuration qui ne peuvent pas être définis via des variables d'environnement spécifiques à un langage.

> **Note**
>
> Les invocations peuvent échouer si le script encapsuleur ne démarre pas correctement le processus de runtime.

Les scripts encapsulateurs sont pris en charge sur tous les environnements d'exécution Lambda natifs. Les scripts Wrapper ne sont pas pris en charge sur Exécutions uniquement basées sur le système d'exploitation (famille d'environnements d'exécution `provided`).

Lorsque vous utilisez un script encapsuleur pour votre fonction, Lambda démarre le runtime en utilisant votre script. Lambda envoie à votre script le chemin d'accès de l'interpréteur et tous les arguments d'origine pour le démarrage du runtime standard. Votre script peut étendre ou transformer le comportement de démarrage du programme. Par exemple, le script peut injecter et modifier des arguments, définir des variables d'environnement ou capturer des métriques, des erreurs et d'autres informations de diagnostic.

Vous spécifiez le script en définissant la valeur de la variable d'environnement `AWS_LAMBDA_EXEC_WRAPPER` comme étant le chemin vers le système de fichiers d'un fichier binaire ou d'un script exécutable.

Exemple : Créer et utiliser un script encapsuleur en tant que couche Lambda

Dans l'exemple suivant, vous créez un script encapsuleur pour démarrer l'interpréteur Python avec l'option `-X importtime`. Lorsque vous exécutez la fonction, Lambda génère une entrée de journal pour indiquer la durée de chaque importation.

Pour créer et utiliser un script encapsuleur en tant que couche

1. Créez un répertoire pour la couche :

   ```
   mkdir -p python-wrapper-layer/bin
   cd python-wrapper-layer/bin
   ```

2. Dans le répertoire `bin`, collez le code suivant dans un nouveau fichier nommé `importtime_wrapper`. Il s'agit du script encapsuleur.

   ```
   #!/bin/bash
   ```

```
# the path to the interpreter and all of the originally intended arguments
args=("$@")

# the extra options to pass to the interpreter
extra_args=("-X" "importtime")

# insert the extra options
args=("${args[@]:0:$#-1}" "${extra_args[@]}" "${args[@]: -1}")

# start the runtime with the extra options
exec "${args[@]}"
```

3. Fournissez au script des autorisations d'exécution :

```
chmod +x importtime_wrapper
```

4. Créez un fichier .zip pour la couche :

```
cd ..
zip -r ../python-wrapper-layer.zip .
```

5. Vérifiez que votre fichier .zip possède la structure de répertoire suivante :

```
python-wrapper-layer.zip
# bin
    # importtime_wrapper
```

6. Créez une couche à l'aide du package .zip.
7. Créez une fonction à l'aide de la console Lambda.
   a. Ouvrez la console Lambda.
   b. Choisissez Créer une fonction.
   c. Indiquez un Function name (Nom de fonction).
   d. Pour Environnement d'exécution, choisissez le dernier environnement d'exécution Python pris en charge (Dernier pris en charge).
   e. Choisissez Créer une fonction.
8. Ajoutez la couche à votre fonction.
   a. Choisissez votre fonction, puis choisissez l'onglet Code s'il n'est pas déjà sélectionné.
   b. Faites défiler jusqu'à la section Couches, puis choisissez Ajouter une couche.

c. Pour Source de la couche, sélectionnez Couches personnalisées, puis choisissez votre couche dans la liste déroulante Couches personnalisées.

d. Pour Version, choisissez 1.

e. Choisissez Ajouter.

9. Ajoutez la variable d'environnement encapsuleur.

a. Choisissez Configuration, puis Variables d'environnement.

b. Sous Variables d'environnement, choisissez Modifier.

c. Choisissez Ajouter une variable d'environnement.

d. Pour Clé, entrez `AWS_LAMBDA_EXEC_WRAPPER`.

e. Dans Valeur, saisissez `/opt/bin/importtime_wrapper` (`/opt/` + la structure de dossiers de votre couche .zip).

f. Choisissez Enregistrer.

10. Testez le script encapsuleur.

a. Choisissez l'onglet Test.

b. Sous Événement de test, choisissez Tester. Il n'est pas nécessaire de créer un événement de test, l'événement par défaut fonctionnera.

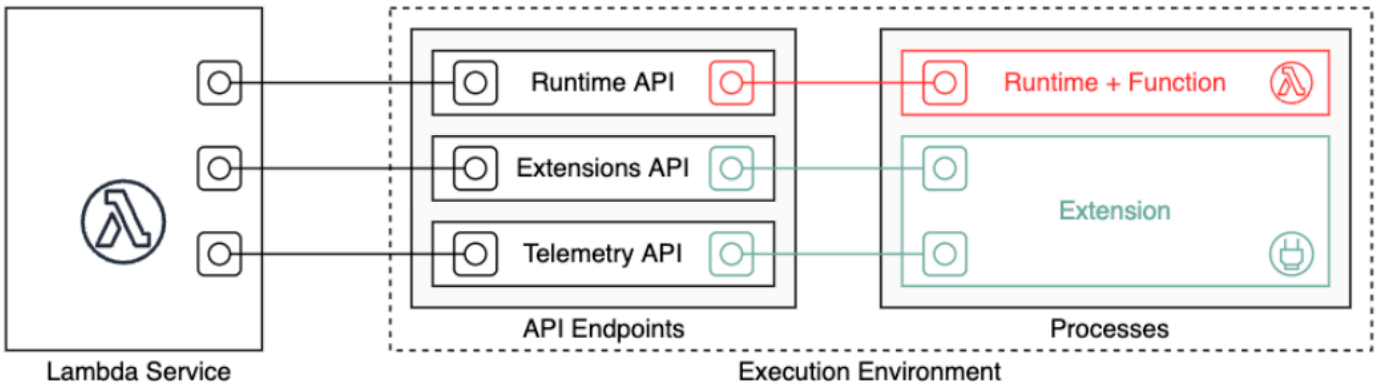
c. Faites défiler la page jusqu'à Sortie de journal. Votre script encapsuleur ayant lancé l'interpréteur Python avec l'option `-X importtime`, les journaux indiquent le temps requis pour chaque importation. Exemples :

```
532 |           collections
import time:        63 |         63 |           _functools
import time:      1053 |       3646 |         functools
import time:      2163 |       7499 |       enum
import time:       100 |        100 |         _sre
import time:       446 |        446 |           re._constants
import time:       691 |       1136 |         re._parser
import time:       378 |        378 |         re._casefix
import time:       670 |       2283 |       re._compiler
import time:       416 |        416 |       copyreg
```

Utilisation de l'API de l'environnement d'exécution Lambda pour des environnements d'exécution personnalisés

AWS Lambda fournit une API HTTP permettant aux environnements d'exécution personnalisés de recevoir des événements d'appel de Lambda et de renvoyer les données de réponse dans l'environnement d'exécution Lambda. Cette section contient la référence de l'API de l'environnement d'exécution Lambda.

> **Les instances gérées Lambda prennent en charge les demandes simultanées**
>
> Les instances gérées Lambda utilisent la même API d'exécution que les fonctions Lambda (par défaut). La principale différence réside dans le fait que les instances gérées peuvent accepter `/next` des `/response` demandes simultanées et jusqu'à la `AWS_LAMBDA_MAX_CONCURRENCY` limite configurée. Cela permet de traiter plusieurs appels simultanément dans un environnement d'exécution unique. Pour plus d'informations sur les instances gérées, consultezComprendre l'environnement d'exécution des instances gérées Lambda.



La spécification OpenAPI pour la version d'API de l'exécution 2018-06-01 est disponible dans runtime-api.zip

Pour créer une URL de requête d'API, les exécutions obtiennent le point de terminaison de l'API à partir de la variable d'environnement `AWS_LAMBDA_RUNTIME_API` et ajoutent la version de l'API ainsi que le chemin d'accès de ressource souhaité.

Example Demande

```
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next"
```

Méthodes d'API

- Invocation suivante
- Réponse d'invocation
- Erreur d'initialisation
- Erreur d'invocation

Invocation suivante

Chemin – `/runtime/invocation/next`

Méthode – GET

L'exécution envoie ce message à Lambda pour demander un événement d'invocation. Le corps de la réponse contient la charge utile provenant de l'invocation, qui est un document JSON contenant les données d'événements du déclencheur de la fonction. Les en-têtes de la réponse contiennent des données supplémentaires sur l'invocation.

En-têtes de réponse

- `Lambda-Runtime-Aws-Request-Id` – ID de demande qui identifie la demande ayant déclenché l'invocation de la fonction.

  Par exemple, `8476a536-e9f4-11e8-9739-2dfe598c3fcd`.
- `Lambda-Runtime-Deadline-Ms` – Date à laquelle la fonction expire, exprimée en millisecondes au format horaire Unix.

  Par exemple, `1542409706888`.
- `Lambda-Runtime-Invoked-Function-Arn` – ARN de la fonction Lambda, de la version ou de l'alias spécifiés dans l'invocation.

  Par exemple, `arn:aws:lambda:us-east-2:123456789012:function:custom-runtime`.
- `Lambda-Runtime-Trace-Id` – En-tête de suivi AWS X-Ray.

Par exemple, `Root=1-5bef4de7-ad49b0e87f6ef6c87fc2e700;Parent=9a9197af755a6419;Sampled=1`.

- `Lambda-Runtime-Client-Context`— Pour les appels depuis le SDK AWS mobile, les données relatives à l'application client et à l'appareil.
- `Lambda-Runtime-Cognito-Identity`— Pour les appels depuis le SDK AWS mobile, les données relatives au fournisseur d'identité Amazon Cognito.

Ne définissez pas de délai d'expiration pour la demande `GET`, car la réponse peut être retardée. Entre le moment où Lambda amorce le runtime et celui où le runtime a un événement à renvoyer, le processus d'exécution peut être bloqué pendant plusieurs secondes.

L'ID de demande suit l'invocation au sein de Lambda. Utilisez-le pour spécifier l'invocation lorsque vous envoyez la réponse.

L'en-tête de suivi contient l'ID de suivi, l'ID parent et la décision d'échantillonnage. Si la demande est échantillonnée, elle a été échantillonnée par Lambda ou un service en amont. Le runtime doit définir l'`_X_AMZN_TRACE_ID` avec la valeur de l'en-tête. Le SDK X-Ray le lit pour obtenir IDs et déterminer s'il convient de suivre la demande.

Réponse d'invocation

Chemin – `/runtime/invocation/AwsRequestId/response`

Méthode – POST

Une fois l'exécution de la fonction terminée, le runtime envoie une réponse à l'invocation à Lambda. Pour les invocations synchrones, Lambda envoie la réponse au client.

Example demande d'opération réussie

```
REQUEST_ID=156cb537-e2d4-11e8-9b34-d36013741fb9
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/
response"  -d "SUCCESS"
```

Erreur d'initialisation

Si la fonction renvoie une erreur ou si le runtime rencontre une erreur lors de l'initialisation, le runtime utilise cette méthode pour signaler l'erreur à Lambda.

Chemin – `/runtime/init/error`

Méthode – POST

En-têtes

`Lambda-Runtime-Function-Error-Type` – Type d'erreur que l'environnement d'exécution a rencontré. Requis : non.

Cet en-tête se compose d'une valeur de chaîne. Lambda accepte n'importe quelle chaîne, mais nous recommandons le format <category.reason>. Par exemple :

- Durée d'exécution. NoSuchHandler
- Durée d'exécution. APIKeyNotFound
- Durée d'exécution. ConfigInvalid
- Durée d'exécution. UnknownReason

Paramètres de corps

`ErrorRequest` : informations sur l'erreur. Requis : non.

Ce champ est un objet JSON avec la structure suivante :

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

Notez que Lambda accepte n'importe quelle valeur pour `errorType`.

L'exemple suivant montre un message d'erreur de fonction Lambda indiquant que la fonction n'a pas pu analyser les données d'événement fournies dans l'invocation.

Example Erreur de fonction

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
```

```
}
```

Paramètres du corps de la réponse

- `StatusResponse` – String. Informations d'état, envoyées avec les codes de réponse 202.
- `ErrorResponse`— Informations d'erreur supplémentaires, envoyées avec les codes de réponse d'erreur. ErrorResponse contient un type d'erreur et un message d'erreur.

Codes de réponse

- 202 – Accepté
- 403 – Interdit
- 500 – Erreur de conteneur. État non récupérable. L'environnement d'exécution doit se terminer rapidement.

Example demande d'erreur d'initialisation

```
ERROR="{\"errorMessage\" : \"Failed to load function.\", \"errorType\" :
 \"InvalidFunctionException\"}"
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/init/error" -d "$ERROR" --
header "Lambda-Runtime-Function-Error-Type: Unhandled"
```

Erreur d'invocation

Si la fonction renvoie une erreur ou si le runtime rencontre une erreur, le runtime utilise cette méthode pour signaler l'erreur à Lambda.

Chemin – `/runtime/invocation/`*`AwsRequestId`*`/error`

Méthode – POST

En-têtes

`Lambda-Runtime-Function-Error-Type` – Type d'erreur que l'environnement d'exécution a rencontré. Requis : non.

Cet en-tête se compose d'une valeur de chaîne. Lambda accepte n'importe quelle chaîne, mais nous recommandons le format <category.reason>. Par exemple :

- Durée d'exécution. NoSuchHandler
- Durée d'exécution. APIKeyNotFound
- Durée d'exécution. ConfigInvalid
- Durée d'exécution. UnknownReason

Paramètres de corps

`ErrorRequest` : informations sur l'erreur. Requis : non.

Ce champ est un objet JSON avec la structure suivante :

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

Notez que Lambda accepte n'importe quelle valeur pour `errorType`.

L'exemple suivant montre un message d'erreur de fonction Lambda indiquant que la fonction n'a pas pu analyser les données d'événement fournies dans l'invocation.

Example Erreur de fonction

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

Paramètres du corps de la réponse

- `StatusResponse` – String. Informations d'état, envoyées avec les codes de réponse 202.
- `ErrorResponse`— Informations d'erreur supplémentaires, envoyées avec les codes de réponse d'erreur. ErrorResponse contient un type d'erreur et un message d'erreur.

Codes de réponse

- 202 – Accepté

- 400 – Demande erronée.
- 403 – Interdit
- 500 – Erreur de conteneur. État non récupérable. L'environnement d'exécution doit se terminer rapidement.

Example demande d'erreur

```
REQUEST_ID=156cb537-e2d4-11e8-9b34-d36013741fb9
ERROR="{\"errorMessage\" : \"Error parsing event data.\", \"errorType\" :
 \"InvalidEventDataException\"}"
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/error"
 -d "$ERROR" --header "Lambda-Runtime-Function-Error-Type: Unhandled"
```

Quand utiliser les environnements d'exécution réservés aux systèmes d'exploitation de Lambda

Lambda fournit des environnements d'exécution gérés pour Java, Python, Node.js, .NET et Ruby. Pour créer des fonctions Lambda dans un langage de programmation qui n'est pas disponible en tant qu'environnement d'exécution géré, utilisez un environnement d'exécution uniquement basé sur le système d'exploitation (famille des environnements d'exécution `provided`). Il existe trois principaux cas d'utilisation pour les environnements d'exécution uniquement basés sur le système d'exploitation :

- Compilation native ahead-of-time (AOT) : des langages tels que Go, Rust, Swift et C++ se compilent nativement en un binaire exécutable, qui ne nécessite pas d'exécution de langage dédié. Ces langages ont uniquement besoin d'un environnement de système d'exploitation dans lequel le binaire compilé peut s'exécuter. Vous pouvez également utiliser des environnements d'exécution uniquement pour le système d'exploitation Lambda pour déployer des fichiers binaires compilés avec .NET Native AOT et Java GraalVM Native Image.

  Vous devez inclure un client d'interface d'exécution. Le client de l'interface d'exécution appelle l'Utilisation de l'API de l'environnement d'exécution Lambda pour des environnements d'exécution personnalisés pour récupérer les invocations de fonction, puis appelle votre gestionnaire de fonctions. Lambda fournit des clients d'interface d'exécution pour Rust, Go, .NET Native AOT, Swift (expérimental) et C++ (expérimental).

  Vous devez compiler votre binaire pour un environnement Linux et pour la même architecture de jeu d'instructions que celle que vous prévoyez d'utiliser pour la fonction (x86_64 ou arm64).
- Runtimes tiers : vous pouvez exécuter des fonctions Lambda à off-the-shelf l'aide d'environnements d'exécution tels que Bref pour PHP.
- Exécutions personnalisées : vous pouvez créer votre propre environnement d'exécution pour une langue ou une version de langue pour laquelle Lambda ne fournit pas d'environnement d'exécution géré, comme Node.js 19. Pour de plus amples informations, veuillez consulter Création d'un environnement d'exécution personnalisé pour AWS Lambda. Il s'agit du cas d'utilisation le moins courant pour les environnements d'exécution uniquement basés sur le système d'exploitation.

Lambda prend en charge les environnements d'exécution uniquement basés sur le système d'exploitation suivants :

Name	Identifiant	Système d'exploitation	Date d'obsolescence	Créer la fonction de blocage	Mettre à jour la fonction de blocage
Exécution réservée au système d'exploitation	`provided.al2023`	Amazon Linux 2	30 juin 2029	31 juillet 2029	31 août 2029
Exécution réservée au système d'exploitation	`provided.al2`	Amazon Linux 2	31 juillet 2026	31 août 2026	30 sept. 2026

L'environnement d'exécution Amazon Linux 2023 (`provided.al2023`) offre plusieurs avantages par rapport à Amazon Linux 2, notamment un encombrement de déploiement réduit et des versions mises à jour de bibliothèques telles que `glibc`.

L'environnement d'exécution `provided.al2023` utilise `dnf` comme gestionnaire de packages au lieu de `yum`, qui est le gestionnaire de packages par défaut dans Amazon Linux 2. Pour plus d'informations sur les différences entre `provided.al2023` et`provided.al2`, consultez Présentation du runtime Amazon Linux 2023 AWS Lambda sur le blog AWS Compute.

Création d'un environnement d'exécution personnalisé pour AWS Lambda

Vous pouvez implémenter un AWS Lambda environnement d'exécution dans n'importe quel langage de programmation. Un environnement d'exécution est un programme qui exécute la méthode de gestionnaire d'une fonction Lambda lors de l'invocation de la fonction. Vous pouvez inclure le runtime dans le package de déploiement de votre fonction ou le distribuer dans une couche. Lorsque vous créez la fonction Lambda, choisissez un environnement d'exécution uniquement pour le système d'exploitation (la famille d'exécution `provided`).

> **Note**
>
> La création d'un environnement d'exécution personnalisé est un cas d'utilisation avancé. Si vous recherchez des informations sur la compilation vers un binaire natif ou sur

l'utilisation d'un off-the-shelf environnement d'exécution tiers, consultezQuand utiliser les environnements d'exécution réservés aux systèmes d'exploitation de Lambda.

Pour une présentation détaillée du processus de déploiement d'un environnement d'exécution personnalisé, voir Tutoriel : création d'un environnement d'exécution personnalisé.

Rubriques

- Exigences
- Implémentation du flux de réponses dans un environnement d'exécution personnalisé
- Création d'environnements d'exécution personnalisés pour les instances gérées Lambda

Exigences

Les environnements d'exécution personnalisés doivent effectuer certaines tâches d'initialisation et de traitement. Une exécution exécute le code d'installation de la fonction, lit le nom du gestionnaire à partir d'une variable d'environnement et lit les événements d'invocation à partir de l'API d'exécution Lambda. L'environnement d'exécution transmet les données d'événements au gestionnaire de la fonction, et renvoie la réponse du gestionnaire à Lambda.

Tâches d'initialisation

Les tâches d'initialisation sont exécutées une seule fois par instance de la fonction pour préparer l'environnement à la gestion des invocations.

- Récupérer les paramètres – Lecture des variables d'environnement pour obtenir des détails sur la fonction et l'environnement.
  - `_HANDLER` – Emplacement du gestionnaire, issu de la configuration de la fonction. Le format standard est *`file`*`.`*`method`*, où `file` est le nom du fichier sans extension et `method` est le nom d'une méthode ou fonction qui est définie dans le fichier.
  - `LAMBDA_TASK_ROOT` – Répertoire contenant le code de la fonction.
  - `AWS_LAMBDA_RUNTIME_API` – Hôte et port de l'API d'exécution.

  Pour obtenir la liste complète des variables disponibles, consultez Variables d'environnement d'exécution définies.

- Initialiser la fonction – Charger le fichier de gestionnaire et exécuter tout code global ou statique qu'il contient. Les fonctions doivent créer des ressources statiques telles que des clients de kit SDK et des connexions de base de données une seule fois, puis les réutiliser pour plusieurs invocations.
- Gérer les erreurs : si une erreur se produit, appelez l'API erreur de l'initialisation et quittez immédiatement.

L'initialisation est comptabilisée dans le délai d'attente et la durée d'exécution facturés. Lorsqu'une exécution déclenche l'initialisation d'une nouvelle instance de votre fonction, vous pouvez voir le temps d'initialisation dans les journaux et le suivi AWS X-Ray.

Example journal

```
REPORT RequestId: f8ac1208... Init Duration: 48.26 ms   Duration: 237.17 ms   Billed
 Duration: 300 ms   Memory Size: 128 MB   Max Memory Used: 26 MB
```

Traitement des tâches

Pendant son exécution, l'environnement d'exécution utilise l'interface d'environnement d'exécution Lambda pour gérer les événements entrants et signaler des erreurs. Après avoir terminé les tâches d'initialisation, l'environnement d'exécution traite les événements entrants dans une boucle. Dans votre code d'exécution, effectuez les étapes suivantes dans l'ordre.

- Obtenir un événement – Appeler l'API d'invocation suivante pour obtenir l'événement suivant. Le corps de la réponse contient les données de l'événement. Les en-têtes de la réponse contiennent l'ID de la demande et d'autres informations.
- Propager l'en-tête de suivi – Obtenir l'en-tête de suivi X-Ray à partir de l'en-tête `Lambda-Runtime-Trace-Id` dans la réponse de l'API. Définissez la variable d'environnement `_X_AMZN_TRACE_ID` localement avec la même valeur. Le kit SDK X-Ray utilise cette valeur pour connecter les données de suivi entre les services.
- Créer un objet de contexte – Créer un objet avec des informations de contexte à partir des variables d'environnement et des en-têtes de la réponse de l'API.
- Invoquer le gestionnaire de fonctions – Transmettre l'événement et l'objet contexte au gestionnaire.
- Gérer la réponse – Appeler l'API réponse d'invocation pour afficher la réponse du gestionnaire.
- Gérer les erreurs : si une erreur se produit, appelez l'API erreur de l'invocation.
- Nettoyage : libérer les ressources inutilisées, envoyer des données à d'autres services ou accomplir des tâches supplémentaires avant de passer à l'événement suivant.

Entrypoint

Le point d'entrée d'un runtime personnalisé est un fichier exécutable nommé `bootstrap`. Le fichier d'amorçage peut être l'environnement d'exécution, ou il peut invoquer un autre fichier qui crée le runtime. Si la racine de votre package de déploiement ne contient pas de fichier nommé `bootstrap`, Lambda recherche le fichier dans les couches de la fonction. Si le fichier `bootstrap` n'existe pas ou n'est pas exécutable, votre fonction renvoie une erreur `Runtime.InvalidEntrypoint` au moment de l'invocation.

Voici un exemple de `bootstrap` fichier qui utilise une version groupée de Node.js pour exécuter un JavaScript environnement d'exécution dans un fichier distinct nommé`runtime.js`.

Example bootstrap

```
#!/bin/sh
    cd $LAMBDA_TASK_ROOT
    ./node-v11.1.0-linux-x64/bin/node runtime.js
```

Implémentation du flux de réponses dans un environnement d'exécution personnalisé

Pour les fonctions de streaming de réponses, les points de terminaison `response` et `error` ont un comportement légèrement modifié qui permet à l'exécution de diffuser des réponses partielles au client et de renvoyer les charges utiles par morceaux. Pour plus d'informations sur le comportement spécifique, consultez ce qui suit :

- `/runtime/invocation/AwsRequestId/response` : propage l'en-tête `Content-Type` de l'environnement d'exécution pour l'envoyer au client. Lambda renvoie la charge utile de la réponse en morceaux via l'encodage de transfert en bloc HTTP/1.1. Le flux de réponse peut avoir une taille maximale de 20 Mio. Pour envoyer le flux de réponse à Lambda, l'environnement d'exécution doit :
  - Définir l'en-tête HTTP `Lambda-Runtime-Function-Response-Mode` sur `streaming`.
  - Définissez l'en-tête `Transfer-Encoding` sur `chunked`.
  - Écrire la réponse conformément à la spécification d'encodage de transfert en bloc HTTP/1.1.
  - Fermer la connexion sous-jacente après avoir écrit la réponse avec succès.
- `/runtime/invocation/AwsRequestId/error` : l'exécution peut utiliser ce point de terminaison pour signaler des erreurs de fonction ou d'exécution à Lambda, qui accepte également l'en-tête `Transfer-Encoding`. Ce point de terminaison ne peut être appelé qu'avant que l'environnement d'exécution ne commence à envoyer une réponse à l'invocation.

- Signaler les erreurs en cours de diffusion à l'aide des en-têtes de suivi d'erreur dans `/runtime/invocation/AwsRequestId/response`. Pour signaler les erreurs qui se produisent après que l'exécution commence à écrire la réponse à l'invocation, l'exécution peut optionnellement attacher des en-têtes de fin HTTP nommés `Lambda-Runtime-Function-Error-Type` et `Lambda-Runtime-Function-Error-Body`. Lambda considère cette réponse comme une réponse réussie et transmet les métadonnées d'erreur que l'exécution fournit au client.

> **Note**
>
> Pour joindre des en-têtes de fin, l'environnement d'exécution doit définir la valeur de l'en-tête `Trailer` au début de la demande HTTP. C'est une exigence de la spécification de l'encodage de transfert en bloc HTTP/1.1.

  - `Lambda-Runtime-Function-Error-Type` : le type d'erreur que l'exécution a rencontré. Cet en-tête se compose d'une valeur de chaîne. Lambda accepte n'importe quelle chaîne, mais nous recommandons un format de. *`<category.reason>`* Par exemple, `Runtime.APIKeyNotFound`.
  - `Lambda-Runtime-Function-Error-Body` : informations encodées en Base64 sur l'erreur.

Création d'environnements d'exécution personnalisés pour les instances gérées Lambda

Les instances gérées Lambda utilisent la même API d'exécution que les fonctions Lambda (par défaut). Cependant, il existe des différences majeures dans la manière dont les environnements d'exécution personnalisés doivent être mis en œuvre pour prendre en charge le modèle d'exécution simultanée des instances gérées.

Gestion des demandes simultanées

La principale différence lors de la création d'environnements d'exécution personnalisés pour les instances gérées est la prise en charge des appels simultanés. Contrairement aux fonctions Lambda (par défaut) où le moteur d'exécution traite un appel à la fois, les instances gérées peuvent traiter plusieurs appels simultanément au sein d'un même environnement d'exécution.

Votre environnement d'exécution personnalisé doit :

- Support des **`/next`** demandes simultanées : le moteur d'exécution peut effectuer plusieurs appels simultanés à l'API d'appel suivante, jusqu'à la limite spécifiée par la variable d'`AWS_LAMBDA_MAX_CONCURRENCY`environnement.
- Gérer les **`/response`** demandes simultanées : plusieurs invocations peuvent appeler simultanément l'API de réponse à l'appel.
- Mettez en œuvre une gestion des demandes adaptée aux threads : assurez-vous que les appels simultanés n'interfèrent pas les uns avec les autres en gérant correctement les ressources et les états partagés.
- Utiliser une demande unique IDs : suivez chaque appel séparément à l'aide de l'`Lambda-Runtime-Aws-Request-Id`en-tête pour faire correspondre les réponses aux demandes correspondantes.

Modèle de mise en œuvre

Un modèle d'implémentation typique pour les environnements d'exécution des instances gérées implique la création de threads de travail ou de processus pour gérer les invocations simultanées :

1. Lire la limite de simultanéité : lors de l'initialisation, lisez la variable d'`AWS_LAMBDA_MAX_CONCURRENCY`environnement pour déterminer le nombre d'appels simultanés à prendre en charge.
2. Créer un pool de travailleurs : initialisez un pool de travailleurs (threads, processus ou tâches asynchrones) égal à la limite de simultanéité.
3. Boucle de traitement des travailleurs — Chaque travailleur indépendamment :
   - Appels `/runtime/invocation/next` pour obtenir un événement d'invocation
   - Invoque le gestionnaire de fonctions avec les données de l'événement
   - Publie la réponse sur `/runtime/invocation/AwsRequestId/response`
   - Répète la boucle

Considérations supplémentaires

- Format de journalisation : les instances gérées ne prennent en charge que le format de journal JSON. Assurez-vous que votre environnement d'exécution respecte la variable d'`AWS_LAMBDA_LOG_FORMAT`environnement et utilise uniquement le format JSON. Pour de plus amples informations, veuillez consulter Configuration des formats de journal JSON et en texte brut.

- Ressources partagées : soyez prudent lorsque vous utilisez des ressources partagées telles que le `/tmp` répertoire avec des invocations simultanées. Mettez en place des mécanismes de verrouillage appropriés pour éviter les conditions de course.

Pour plus d'informations sur les environnements d'exécution des instances gérées Lambda, consultez. Comprendre l'environnement d'exécution des instances gérées Lambda

Tutoriel : création d’un environnement d’exécution personnalisé

Dans ce didacticiel, vous allez créer une fonction Lambda avec un runtime personnalisé. Vous commencez par inclure le runtime dans le package de déploiement de la fonction. Ensuite, vous le migrez vers une couche que vous gérez indépendamment de la fonction. Enfin, vous partagez la couche du runtime en mettant à jour sa stratégie d’autorisations basée sur les ressources.

Prérequis

Ce didacticiel suppose que vous avez quelques connaissances des opérations Lambda de base et de la console Lambda. Si ce n’est déjà fait, suivez les instructions fournies dans Créer une fonction Lambda à l'aide de la console pour créer votre première fonction Lambda.

Pour effectuer les étapes suivantes, vous avez besoin de l’AWS CLI version 2. Les commandes et la sortie attendue sont répertoriées dans des blocs distincts :

```
aws --version
```

Vous devriez voir la sortie suivante:

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

Pour les commandes longues, un caractère d’échappement (\) est utilisé pour les fractionner en plusieurs lignes.

Sur Linux et macOS, utilisez votre gestionnaire de shell et de package préféré.

> **Note**
>
> Sous Windows, certaines commandes CLI Bash que vous utilisez couramment avec Lambda (par exemple `zip`) ne sont pas prises en charge par les terminaux intégrés du système

d'exploitation. Installez le sous-système Windows pour Linux afin d'obtenir une version intégrée à Windows d'Ubuntu et Bash. Les exemples de commandes CLI de ce guide utilisent le formatage Linux. Les commandes qui incluent des documents JSON en ligne doivent être reformatées si vous utilisez la CLI Windows.

Vous avez besoin d'un rôle IAM pour créer une fonction Lambda. Le rôle a besoin d'une autorisation pour envoyer des journaux à CloudWatch Logs et accéder aux Services AWS que votre fonction utilise. Si vous ne possédez pas de rôle pour le développement de fonction, créez-en un.

Pour créer un rôle d'exécution

1. Ouvrez la page Roles (Rôles) dans la console IAM.
2. Sélectionnez Créer un rôle.
3. Créez un rôle avec les propriétés suivantes :

   - Entité de confiance – Lambda.
   - Autorisations – AWSLambdaBasicExecutionRole.
   - Nom de rôle – **`lambda-role`**.

   La stratégie AWSLambdaBasicExecutionRole possède les autorisations dont la fonction a besoin pour écrire des journaux dans CloudWatch Logs.

Créer une fonction

Créez une fonction Lambda avec un runtime personnalisé. Cet exemple comprend deux fichiers : un fichier `bootstrap` d'exécution et un gestionnaire de fonctions. Tous deux sont mis en œuvre en Bash.

1. Créez un répertoire pour le projet, puis passez à ce répertoire.

   ```
   mkdir runtime-tutorial
   cd runtime-tutorial
   ```

2. Créez un nouveau fichier appelé `bootstrap`. Il s'agit de l'environnement d'exécution personnalisé.

Example amorçage

```
#!/bin/sh

set -euo pipefail

# Initialization - load function handler
source $LAMBDA_TASK_ROOT/"$(echo $_HANDLER | cut -d. -f1).sh"

# Processing
while true
do
  HEADERS="$(mktemp)"
  # Get an event. The HTTP request will block until one is received
  EVENT_DATA=$(curl -sS -LD "$HEADERS" "http://
${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next")

  # Extract request ID by scraping response headers received above
  REQUEST_ID=$(grep -Fi Lambda-Runtime-Aws-Request-Id "$HEADERS" | tr -d
 '[:space:]' | cut -d: -f2)

  # Run the handler function from the script
  RESPONSE=$($(echo "$_HANDLER" | cut -d. -f2) "$EVENT_DATA")

  # Send the response
  curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/
response"  -d "$RESPONSE"
done
```

Le runtime charge un script de fonction à partir du package de déploiement. Il utilise deux variables pour localiser le script. `LAMBDA_TASK_ROOT` lui indique où le package a été extrait et `_HANDLER` inclut le nom du script.

Une fois que l'environnement d'exécution charge le script de la fonction, il utilise l'API d'exécution pour récupérer un événement d'invocation à partir de Lambda, transmet l'événement au gestionnaire, puis renvoie la réponse à Lambda. Pour obtenir l'ID de la demande, le runtime enregistre les en-têtes à partir de la réponse de l'API dans un fichier temporaire et lit l'en-tête `Lambda-Runtime-Aws-Request-Id` à partir du fichier.

> **Note**
>
> Les runtimes ont d'autres responsabilités, notamment la gestion des erreurs et la fourniture d'informations de contexte au gestionnaire. Pour plus d'informations, consultez Exigences.

3. Créez un script pour la fonction. L'exemple de script suivant définit une fonction de gestionnaire qui accepte les données des événements, la consigne dans `stderr`, puis la renvoie.

   Example function.sh

   ```
   function handler () {
     EVENT_DATA=$1
     echo "$EVENT_DATA" 1>&2;
     RESPONSE="Echoing request: '$EVENT_DATA'"

     echo $RESPONSE
   }
   ```

   Le répertoire `runtime-tutorial` devrait maintenant ressembler à ceci :

   ```
   runtime-tutorial
   # bootstrap
   # function.sh
   ```

4. Rendez les fichiers exécutables et ajoutez-les dans une archive ZIP. Vous obtiendrez alors le package de déploiement.

   ```
   chmod 755 function.sh bootstrap
   zip function.zip function.sh bootstrap
   ```

5. Créez une fonction nommée `bash-runtime`. Pour `--role`, entrez l'ARN de votre rôle d'exécution Lambda.

   ```
   aws lambda create-function --function-name bash-runtime \
   --zip-file fileb://function.zip --handler function.handler --runtime
    provided.al2023 \
   --role arn:aws:iam::123456789012:role/lambda-role
   ```

6. Invoquer la fonction.

```
aws lambda invoke --function-name bash-runtime --payload '{"text":"Hello"}'
 response.txt --cli-binary-format raw-in-base64-out
```

L'option cli-binary-format est obligatoire si vous utilisez AWS CLI version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales prises en charge par l'AWS CLI dans le Guide de l'utilisateur AWS Command Line Interface version 2.

Vous devriez obtenir une réponse comme celle-ci :

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

7. Vérifiez la réponse.

```
cat response.txt
```

Vous devriez obtenir une réponse comme celle-ci :

```
Echoing request: '{"text":"Hello"}'
```

Créer une couche

Pour séparer le code du runtime du code de la fonction, créez une couche qui contient uniquement le runtime. Les couches vous permettent de développer les dépendances de votre fonction de manière indépendante et peuvent réduire l'utilisation du stockage lorsque vous utilisez la même couche avec plusieurs fonctions. Pour de plus amples informations, consultez Gestion des dépendances Lambda à l'aide de couches.

1. Créez un fichier .zip contient le fichier `bootstrap`.

```
zip runtime.zip bootstrap
```

2. Créez une couche avec la commande publish-layer-version.

```
aws lambda publish-layer-version --layer-name bash-runtime --zip-file fileb://
runtime.zip
```

Cela crée la première version de la couche.

Mettre à jour la fonction

Pour utiliser la couche d'exécution dans la fonction, configurez la fonction pour utiliser la couche et supprimez le code de l'environnement d'exécution de la fonction.

1. Mettez à jour la configuration de la fonction pour extraire la couche.

   ```
   aws lambda update-function-configuration --function-name bash-runtime \
   --layers arn:aws:lambda:us-east-1:123456789012:layer:bash-runtime:1
   ```

   Cela ajoute le runtime à la fonction dans le répertoire `/opt`. Pour garantir que Lambda utilise l'environnement d'exécution dans la couche, vous devez supprimer `boostrap` du package de déploiement de la fonction, comme indiqué dans les deux étapes suivantes.

2. Créez un fichier .zip reprenant le code de la fonction.

   ```
   zip function-only.zip function.sh
   ```

3. Mettez à jour le code de la fonction de façon à inclure uniquement le script du gestionnaire.

   ```
   aws lambda update-function-code --function-name bash-runtime --zip-file fileb://
   function-only.zip
   ```

4. Invoquez la fonction pour confirmer qu'elle fonctionne avec la couche de l'environnement d'exécution.

   ```
   aws lambda invoke --function-name bash-runtime --payload '{"text":"Hello"}'
    response.txt --cli-binary-format raw-in-base64-out
   ```

   L'option cli-binary-format est obligatoire si vous utilisez AWS CLI version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande

globales prises en charge par l'AWS CLI dans le Guide de l'utilisateur AWS Command Line Interface version 2.

Vous devriez obtenir une réponse comme celle-ci :

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

5. Vérifiez la réponse.

```
cat response.txt
```

Vous devriez obtenir une réponse comme celle-ci :

```
Echoing request: '{"text":"Hello"}'
```

Mettre à jour le runtime

1. Pour enregistrer des informations sur l'environnement d'exécution, mettez à jour le script du runtime pour générer les variables d'environnement.

Example amorçage

```
#!/bin/sh

set -euo pipefail

# Configure runtime to output environment variables
echo "##  Environment variables:"
env

# Load function handler
source $LAMBDA_TASK_ROOT/"$(echo $_HANDLER | cut -d. -f1).sh"

# Processing
while true
do
  HEADERS="$(mktemp)"
```

```
  # Get an event. The HTTP request will block until one is received
  EVENT_DATA=$(curl -sS -LD "$HEADERS" "http://
${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next")

  # Extract request ID by scraping response headers received above
  REQUEST_ID=$(grep -Fi Lambda-Runtime-Aws-Request-Id "$HEADERS" | tr -d
 '[:space:]' | cut -d: -f2)

  # Run the handler function from the script
  RESPONSE=$($(echo "$_HANDLER" | cut -d. -f2) "$EVENT_DATA")

  # Send the response
  curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/
response"  -d "$RESPONSE"
done
```

2. Créez un fichier .zip reprenant la nouvelle version du fichier `bootstrap`.

```
zip runtime.zip bootstrap
```

3. Créer une nouvelle version de la couche `bash-runtime`.

```
aws lambda publish-layer-version --layer-name bash-runtime --zip-file fileb://
runtime.zip
```

4. Configurez la fonction pour utiliser la nouvelle version de la couche.

```
aws lambda update-function-configuration --function-name bash-runtime \
--layers arn:aws:lambda:us-east-1:123456789012:layer:bash-runtime:2
```

Partager la couche

Pour partager une couche avec un autre Compte AWS, ajoutez une instruction d'autorisations entre comptes à la politique basée sur les ressources de la couche. Exécutez la commande add-layer-version-permission et spécifiez l'ID du compte en tant que `principal`. Dans chaque instruction, vous pouvez accorder une autorisation à un compte unique, à tous les comptes ou à une organisation dans AWS Organizations.

L'exemple suivant autorise le compte 111122223333 à accéder à la version 2 de la couche `bash-runtime`.

```
aws lambda add-layer-version-permission \
  --layer-name bash-runtime \
  --version-number 2 \
  --statement-id xaccount \
  --action lambda:GetLayerVersion \
  --principal 111122223333 \
  --output text
```

Vous devez voir des résultats similaires à ce qui suit :

```
{"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:GetLayerVersion","Resource":"arn:aws:
east-1:123456789012:layer:bash-runtime:2"}
```

Les autorisations ne s'appliquent qu'à une seule version de couche. Répétez le processus chaque fois que vous créez une nouvelle version de la couche.

Nettoyage

Supprimez chaque version de la couche.

```
aws lambda delete-layer-version --layer-name bash-runtime --version-number 1
aws lambda delete-layer-version --layer-name bash-runtime --version-number 2
```

Étant donné que la fonction contient une référence à la version 2 de la couche, elle existe toujours dans Lambda. La fonction continue de fonctionner, mais les fonctions ne peuvent plus être configurées pour utiliser la version supprimée. Si vous modifiez la liste des couches sur la fonction, vous devez spécifier une nouvelle version ou omettre la couche supprimée.

Supprimez la fonction à l'aide de la commande delete-function.

```
aws lambda delete-function --function-name bash-runtime
```

Référentiels open source

AWS Lambda fournit une variété d'outils, de bibliothèques et de composants open source pour vous aider à créer, personnaliser et optimiser vos applications sans serveur. Ces ressources incluent des clients d'interface d'exécution, des bibliothèques d'événements, des images de base de conteneurs, des outils de développement et des exemples de projets gérés AWS et disponibles sur GitHub. En

tirant parti de ces référentiels open source, vous pouvez étendre les fonctionnalités de Lambda, créer des environnements d'exécution personnalisés, traiter des événements provenant de différents AWS services et mieux comprendre les performances de votre fonction. Cette page fournit une présentation des principaux projets open source qui soutiennent le développement de Lambda.

Clients d’interface d’exécution

Les clients d'interface d'exécution Lambda (RICs) sont des bibliothèques open source qui implémentent l'API Runtime et gèrent l'interaction entre le code de votre fonction et le service Lambda. Ces clients gèrent la réception des événements d’invocation, la transmission des informations de contexte et le signalement des erreurs.

Les clients d’interface d’exécution utilisés par les environnements d’exécution gérés et les images de base de conteneurs de Lambda sont open source. Lorsque vous créez des environnements d’exécution personnalisés ou que vous étendez des environnements existants, vous pouvez utiliser ces bibliothèques open source pour simplifier la mise en œuvre. Les GitHub référentiels open source suivants contiennent le code source de Lambda : RICs

- Client d’interface d’environnement d’exécution Node.js
- Clients d’interface d’exécution Python
- Client d’interface d’environnement d’exécution Java
- Client d’interface d’environnement d’exécution Ruby
- Client d’interface d’exécution .NET
- Client d'interface d'exécution Rust
- Client d’interface d’environnement d’exécution Go
- Client d'interface Swift Runtime (expérimental)
- Client d'interface d'exécution C++ (expérimental)
- Images de base Lambda

Pour plus d’informations sur l’utilisation de ces clients pour créer une exécution personnalisée, consultez the section called “Création d’un environnement d’exécution personnalisé”.

Bibliothèques d’événements

Les bibliothèques d'événements Lambda fournissent des définitions de type et des utilitaires d'assistance pour le traitement des événements provenant de différents services. AWS Ces

bibliothèques vous aident à analyser et gérer les données d'événements de manière sécurisée, ce qui facilite l'utilisation d'événements provenant de services comme Amazon S3, Amazon DynamoDB et Amazon API Gateway.

Pour les langages compilés, AWS fournit les bibliothèques d'événements suivantes :

- Bibliothèque d'événements Java
- Bibliothèques d'événements .NET
- Bibliothèque d'événements Go
- Bibliothèque d'événements Rust

Pour les langages interprétés comme Node.js, Python et Ruby, les événements peuvent être analysés directement sous forme d'objets JSON sans nécessiter de bibliothèque distincte. Toutefois, les développeurs utilisant Node.js et Python peuvent tirer parti de powertools for AWS Lambda, qui fournit des schémas intégrés pour les AWS événements proposant des indications de type, une validation des données et des fonctionnalités similaires à celles fournies par les bibliothèques de langage compilées.

- Outils électriques pour TypeScript
- Powertools pour Python

Images de base de conteneur

AWS fournit des images de base de conteneurs open source que vous pouvez utiliser comme point de départ pour créer des images de conteneur pour vos fonctions Lambda. Ces images de base incluent le client d'interface d'environnement d'exécution et les autres composants nécessaires pour exécuter vos fonctions dans l'environnement d'exécution Lambda.

Pour plus d'informations sur les images de base disponibles et sur leur utilisation, consultez le référentiel Images de base AWS Lambda et the section called "Images de conteneurs".

Outils de développement

AWS fournit des outils de développement open source supplémentaires pour vous aider à créer et à optimiser vos fonctions Lambda :

Outils électriques pour AWS Lambda

Powertools for AWS Lambda simplifie le développement sans serveur grâce à des utilitaires essentiels pour empêcher le traitement en double, et au traitement par lots pour la gestion de plusieurs enregistrements et à la bibliothèque Kafka destinée aux consommateurs. Ces fonctionnalités vous aident à minimiser la complexité du code et les frais d'exploitation.

Vous pouvez également tirer parti de la validation intégrée du schéma d'événements, de la journalisation et du suivi structurés, ainsi que de l'intégration du magasin de paramètres, qui sont conçus pour accélérer la création de fonctions Lambda prêtes pour la production tout en suivant les AWS meilleures pratiques bien conçues.

GitHub référentiels :

- Python
- TypeScript
- Java
- .NET

Outils de développement Java

- Java Profiler (expérimental) : un outil pour profiler les fonctions Java Lambda.
- Bibliothèques Java : référentiel contenant une collection complète de bibliothèques et d'outils Java pour le développement Lambda, y compris des projets clés tels que des utilitaires de JUnit test et des outils de profilage.
- Conteneur Java sans serveur : bibliothèque qui vous permet d'exécuter des applications Java existantes sur Lambda avec un minimum de modifications.

Outils de développement .NET

Le référentiel AWS Lambda .NET fournit des bibliothèques et des outils .NET pour le développement Lambda, y compris des projets clés tels que les AWS Lambda outils pour la CLI .NET et le serveur .NET Core pour l'hébergement d'applications .NET Core.

Exemples de projets

Découvrez une collection complète d'exemples de projets et d'applications Lambda dans les référentiels Serverless Land. Ces exemples présentent différents cas d'utilisation, modèles d'intégration et bonnes pratiques Lambda pour vous aider à démarrer avec vos applications sans serveur.

Configuration des AWS Lambda fonctions

Découvrez comment configurer les capacités et options de base de votre fonction Lambda à l'aide de l'API ou de la console Lambda.

archives de fichiers .zip

Créez un package de déploiement de fonctions Lambda lorsque vous souhaitez inclure des dépendances, des couches d'exécution personnalisées ou des fichiers autres que le code de votre fonction. Le package de déploiement est une archive de fichier .zip contenant le code de votre fonction et ses dépendances.

Images de conteneur

Utilisez des images de conteneur pour empaqueter le code de votre fonction et ses dépendances lorsque vous avez besoin de mieux contrôler le processus de génération ou si votre fonction nécessite des configurations d'exécution personnalisées. Vous pouvez créer, tester et déployer des fonctions Lambda sous forme d'images de conteneur à l'aide d'outils tels que Docker CLI.

Mémoire

Découvrez comment et quand augmenter la mémoire des fonctions.

Stockage éphémère

Découvrez comment et quand augmenter la capacité de stockage temporaire de votre fonction.

Expiration

Découvrez comment et quand augmenter le délai d’expiration de votre fonction.

Configuration durable

Permettez à vos fonctions Lambda de s'exécuter pendant des périodes prolongées (jusqu'à un an) tout en conservant leur état entre les invocations. Configurez les délais d'exécution, la rétention des états et le comportement de gestion des versions pour les flux de travail dynamiques de longue durée. Les fonctions durables vous permettent de créer des processus complexes en plusieurs étapes capables de suspendre, de reprendre et de gérer automatiquement les interruptions.

Variables d'environnement

Vous pouvez rendre le code de votre fonction portable et garder les secrets de votre code en les stockant dans la configuration de votre fonction à l'aide de variables d'environnement.

Réseaux sortants

Vous pouvez utiliser votre fonction Lambda avec les AWS ressources d'un Amazon VPC. La connexion de votre fonction à un VPC vous permet d'accéder aux ressources d'un sous-réseau privé, telles que les bases de données relationnelles et les caches.

Réseaux entrants

Vous pouvez utiliser un point de terminaison d'un VPC d'interface pour invoquer vos fonctions Lambda sans traverser l'Internet public.

Système de fichiers

Vous pouvez utiliser votre fonction Lambda pour monter un Amazon EFS dans un répertoire local. Un système de fichiers permet à votre code de fonction d'accéder à des ressources partagées et de les modifier en toute sécurité et avec une simultanéité élevée.

Alias

Vous pouvez configurer vos clients pour qu'ils invoquent une version spécifique de la fonction Lambda en utilisant un alias, au lieu de mettre à jour le client.

Versions

En publiant une version de votre fonction, vous pouvez stocker votre code et votre configuration comme une ressource séparée qui ne peut pas être modifiée.

Balises

Utilisez des balises pour activer le contrôle d'accès basé sur les attributs (ABAC), pour organiser vos fonctions Lambda et pour filtrer et générer des rapports sur vos fonctions à l'aide des services ou AWS Billing and AWS Cost Explorer Cost Management.

Streaming des réponses

Vous pouvez configurer votre fonction Lambda URLs pour transmettre les charges utiles de réponse aux clients. Le streaming de réponses peut profiter aux applications sensibles à la latence en améliorant les performances de temps au premier octet (TTFB). En effet, vous pouvez renvoyer des réponses partielles au client dès qu'elles sont disponibles. En outre, vous pouvez utiliser le streaming de réponses pour créer des fonctions qui renvoient des charges utiles plus importantes.

Déploiement des fonctions Lambda comme des archives de fichiers .zip

Lorsque vous créez une fonction Lambda, vous empaquetez le code de votre fonction dans un package de déploiement. Lambda prend en charge deux types de packages de déploiement : les images conteneurs et les archives de fichier .zip. Le flux de travail de création d'une fonction dépend du type de package de déploiement. Pour configurer une fonction définie en tant qu'image de conteneur, consultez the section called “Images de conteneurs”.

Vous pouvez utiliser la console Lambda et l'API Lambda pour créer une fonction définie avec une archive de fichiers .zip. Vous pouvez également charger un fichier .zip mis à jour pour modifier le code de la fonction.

> **Note**
>
> Vous ne pouvez pas modifier le type de package de déploiement (.zip ou image de conteneur) d’une fonction existante. Par exemple, vous ne pouvez pas convertir une fonction d’image de conteneur pour utiliser un fichier d’archive .zip à la place. Vous devez créer une nouvelle fonction.

Rubriques

- Création de la fonction
- Utilisation de l'éditeur de code de la console
- Mise à jour du code de fonction
- Modification de l'environnement d'exécution
- Modification de l'architecture
- Utilisation de l'API Lambda
- Téléchargement du code de votre fonction
- CloudFormation
- Chiffrement des packages de déploiement Lambda au format .zip

Création de la fonction

Lorsque vous créez une fonction définie avec une archive de fichiers .zip, vous choisissez un modèle de code, la version du langage et le rôle d'exécution pour la fonction. Vous ajoutez votre code de fonction après que Lambda a créé la fonction.

Pour créer la fonction

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez Create function (Créer une fonction).
3. Choisissez Créer à partir de zéro ou Utiliser un plan pour créer votre fonction.
4. Sous Informations de base, procédez comme suit :
    a. Pour Function name (Nom de la fonction), saisissez le nom de la fonction. Les noms de fonctions sont limités à 64 caractères.
    b. Pour Runtime, sélectionnez la version du langage à utiliser pour votre fonction.
    c. (Facultatif) Pour Architecture, choisissez l'architecture de l'ensemble des instructions pour utiliser la fonction. L'architecture par défaut est x86_64. Lorsque vous créez le package de déploiement de la fonction, assurez-vous qu'il est compatible avec cette architecture de l'ensemble des instructions.
5. (Facultatif) Sous Permissions (Autorisations), développez Change default execution role (Modifier le rôle d'exécution par défaut). Vous pouvez créer un rôle d'exécution ou utiliser un rôle existant.
6. (Facultatif) Développez Advanced settings (Paramètres avancés). Vous pouvez choisir une Configuration de signature de code pour la fonction. Vous pouvez également configurer un (VPC Amazon) auquel la fonction puisse accéder.
7. Sélectionnez Créer une fonction.

Lambda crée la nouvelle fonction. Vous pouvez désormais utiliser la console pour ajouter le code de fonction et configurer d'autres paramètres et caractéristiques de la fonction. Pour obtenir des instructions de déploiement de code, consultez la page du gestionnaire pour l'exécution utilisée par votre fonction.

Node.js

> Déployer des fonctions Lambda en Node.js avec des archives de fichiers .zip

Python

- Travailler avec des archives de fichiers .zip pour les fonctions Lambda Python

Ruby

- Déployer des fonctions Lambda en Ruby avec des archives de fichiers .zip

Java

- Déployer des fonctions Lambda en Java avec des archives de fichiers .zip ou JAR

Go

- Déployer des fonctions Lambda Go avec des archives de fichiers .zip

C#

- Créez et déployez des fonctions Lambda C# à l'aide des archives de fichiers .zip

PowerShell

- Déployer des fonctions Lambda PowerShell avec des archives de fichiers .zip

Utilisation de l'éditeur de code de la console

La console crée une fonction Lambda avec un seul fichier source. Pour le scripting, vous pouvez modifier ce fichier et ajouter des fichiers à l'aide de l'éditeur de code intégré. Choisissez Save pour enregistrer les changements. Ensuite, pour exécuter votre code, choisissez Test.

Lorsque vous enregistrez votre code de fonction, la console Lambda crée un package de déploiement d'archive de fichiers .zip. Lorsque vous développez votre code de fonction en dehors de la console (à l'aide d'un IDE), vous devez créer un package de déploiement pour charger votre code dans la fonction Lambda.

Mise à jour du code de fonction

Pour le scripting (Node.js, Python et Ruby), vous pouvez modifier votre code de fonction dans l'éditeur intégré. Si ce code est d'une taille supérieure à 3 Mo, si vous devez ajouter des bibliothèques ou dans le cas des langages que l'éditeur ne prend pas en charge (Java, Go, C#), vous devez charger le code de votre fonction en tant qu'archive .zip. Si l'archive de fichier .zip est d'une taille inférieure à 50 Mo, vous pouvez charger l'archive de fichier .zip à partir de votre machine locale. Si le fichier est d'une taille supérieure à 50 Mo, chargez le fichier dans la fonction à partir d'un compartiment Amazon S3.

Pour charger le code de fonction en tant qu'archive .zip

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction à mettre à jour, puis choisissez l'onglet Code.
3. Sous Code source (Source du code), sélectionnez Upload from (Charger depuis).
4. Choisissez .zip file (fichier .zip), puis Upload (Charger).
   - Dans le sélecteur de fichiers, sélectionnez la nouvelle version de l'image et choisissez Open (Ouvrir), puis Save (Enregistrer).
5. (Alternative à l'étape 4) Choisissez Emplacement Amazon S3.
   - Dans la zone de texte, saisissez l'URL du lien S3 de l'archive de fichiers .zip, puis choisissez Enregistrer.

Modification de l'environnement d'exécution

Si vous mettez à jour la configuration de la fonction pour utiliser un nouvel environnement d'exécution, vous devrez peut-être mettre à jour le code de la fonction pour qu'il soit compatible avec la nouvelle version. Si vous mettez à jour la configuration de la fonction pour utiliser une autre exécution, vous devez fournir un nouveau code de fonction compatible avec l'exécution et l'architecture. Pour obtenir des instructions sur comment créer un package de déploiement pour le code de la fonction, consultez la page du gestionnaire de l'exécution utilisé par la fonction.

Les images de base de Node.js 20, Python 3.12, Java 21, .NET 8, Ruby 3.3 et versions ultérieures sont basées sur l'image de conteneur minimale Amazon Linux 2023. Les images de base antérieures utilisaient Amazon Linux 2. AL2Le 023 offre plusieurs avantages par rapport à Amazon Linux 2, notamment un encombrement de déploiement réduit et des versions mises à jour de bibliothèques telles que`glibc`. Pour plus d'informations, consultez Présentation de l'environnement d'exécution Amazon Linux 2023 pour AWS Lambda sur le blog AWS Compute.

Pour modifier l'environnement d'exécution

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction à mettre à jour, puis choisissez l'onglet Code.
3. Faites défiler jusqu'à la section Runtime settings (Paramètres d'exécution), qui se trouve sous l'éditeur de code.
4. Choisissez Modifier.

a. Pour Runtime (Environnement d'exécution), sélectionnez l'identifiant de l'environnement d'exécution.

b. Pour Gestionnaire, spécifiez le nom de fichier et gestionnaire de votre fonction.

c. Pour Architecture, choisissez l'architecture de l'ensemble des instructions pour utiliser votre fonction.

5. Choisissez Enregistrer.

Modification de l'architecture

Avant de pouvoir modifier l'architecture de l'ensemble des instructions, vous devez vous assurer que le code de la fonction est compatible avec l'architecture cible.

Si vous utilisez Node.js, Python ou Ruby et que vous modifiez le code de fonction dans l'éditeur, le code existant peut s'exécuter sans modification.

Toutefois, si vous fournissez le code de fonction en utilisant un package de déploiement d'archives de fichiers .zip, vous devez préparer une nouvelle archive de fichiers .zip compilée et créée correctement pour l'exécution cible et l'architecture de l'ensemble des instructions. Pour obtenir des instructions, veuillez consulter la page du gestionnaire d'exécution de la fonction.

Pour modifier l'architecture de l'ensemble des instructions

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction à mettre à jour, puis choisissez l'onglet Code.
3. Sous Paramètres du runtime, choisissez Modifier.
4. Pour Architecture, choisissez l'architecture de l'ensemble des instructions pour utiliser votre fonction.
5. Choisissez Enregistrer.

Utilisation de l'API Lambda

Pour créer et configurer une fonction qui utilise une archive de fichiers .zip, utilisez les opérations d'API suivantes :

- CreateFunction

- UpdateFunctionCode
- UpdateFunctionConfiguration

Téléchargement du code de votre fonction

Vous pouvez télécharger la version non publiée actuelle (`$LATEST`) de l'archive .zip du code de votre fonction via la console Lambda. Avant de démarrer, vérifiez que vous disposez des autorisations IAM suivantes :

- `iam:GetPolicy`
- `iam:GetPolicyVersion`
- `iam:GetRole`
- `iam:GetRolePolicy`
- `iam:ListAttachedRolePolicies`
- `iam:ListRolePolicies`
- `iam:ListRoles`

Pour télécharger l'archive .zip du code de fonction

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction pour laquelle vous souhaitez télécharger l'archive .zip du code de la fonction.
3. Dans Aperçu de la fonction, cliquez sur le bouton Télécharger, puis choisissez Télécharger l'archive .zip du code de fonction

    - Vous pouvez également choisir Télécharger AWS SAM le fichier pour générer et télécharger un modèle SAM en fonction de la configuration de votre fonction. Vous pouvez aussi choisir Télécharger les deux pour télécharger à la fois l'archive .zip et le modèle SAM.

CloudFormation

Vous pouvez l'utiliser CloudFormation pour créer une fonction Lambda qui utilise une archive de fichiers .zip. Dans votre CloudFormation modèle, la `AWS::Lambda::Function` ressource spécifie la fonction Lambda. Pour une description des propriétés de la `AWS::Lambda::Function` ressource, reportez-vous AWS::Lambda::Functionau guide de l'AWS CloudFormation utilisateur.

Dans la ressource `AWS::Lambda::Function`, définissez les propriétés suivantes pour créer une fonction définie en tant qu'archive de fichiers .zip :

- AWS::Lambda::Function
  - PackageType — Réglé sur`Zip`.
  - Code – Saisissez le nom du compartiment Amazon S3 et le nom du fichier .zip dans les champs `S3Bucket` et `S3Key`. Pour Node.js ou Python, vous pouvez fournir le code source inclus de votre fonction Lambda.
  - Runtime – Définissez la valeur du runtime.
  - Architecture — Définissez la valeur de l'architecture `arm64` pour utiliser le processeur AWS Graviton2. La valeur de l'architecture par défaut est `x86_64`.

Chiffrement des packages de déploiement Lambda au format .zip

Lambda fournit toujours le chiffrement côté serveur au repos pour les packages de déploiement .zip et les détails de configuration des fonctions avec une AWS KMS key. Par défaut, Lambda utilise une Clé détenue par AWS. Si ce comportement par défaut convient à votre flux de travail, vous n'avez rien d'autre à configurer. AWS l'utilisation de cette clé ne vous est pas facturée.

Si vous préférez, vous pouvez plutôt fournir une clé gérée par le AWS KMS client. Vous pouvez procéder ainsi pour contrôler la rotation de la clé KMS ou pour répondre aux exigences de votre organisation en matière de gestion des clés KMS. Lorsque vous utilisez une clé gérée par le client, seuls les utilisateurs de votre compte ayant accès à la clé KMS peuvent visualiser ou gérer le code ou la configuration de la fonction.

Les clés gérées par le client entraînent des AWS KMS frais standard. Pour en savoir plus, consultez Pricing AWS Key Management Service (Tarification).

Création d’une clé gérée par le client

Vous pouvez créer une clé symétrique gérée par le client en utilisant le AWS Management Console, ou le AWS KMS APIs.

Pour créer une clé symétrique gérée par le client

Suivez les étapes de la rubrique Create a symmetric encryption KMS key dans le Guide du développeur AWS Key Management Service .

Permissions

Stratégie de clé

Les stratégies de clé contrôlent l'accès à votre clé gérée par le client. Chaque clé gérée par le client doit avoir exactement une stratégie de clé, qui contient des instructions qui déterminent les personnes pouvant utiliser la clé et comment elles peuvent l'utiliser. Pour plus d'informations, consultez How to change a key policy dans le Guide du développeur AWS Key Management Service .

Lorsque vous utilisez une clé gérée par le client pour chiffrer un package de déploiement .zip, Lambda n'ajoute aucun octroi à la clé. Au lieu de cela, votre politique AWS KMS clé doit autoriser Lambda à appeler les opérations d' AWS KMS API suivantes en votre nom :

- km : GenerateDataKey
- kms:Decrypt

L'exemple de politique de clé suivant permet à toutes les fonctions Lambda du compte 111122223333 d'appeler les AWS KMS opérations requises pour la clé gérée par le client spécifiée :

Example AWS KMS politique clé

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "lambda.amazonaws.com"
            },
            "Action": [
                "kms:GenerateDataKey",
                "kms:Decrypt"
            ],
            "Resource": "arn:aws:kms:us-east-1:111122223333:key/key-id",
            "Condition": {
                "StringLike": {
                "kms:EncryptionContext:aws:lambda:FunctionArn":
 "arn:aws:lambda:us-east-1:111122223333:function:*"
```

```
                }
            }
        }
    ]
}
```

Pour plus d'informations sur le dépannage des clés d'accès, consultez le Guide du développeur AWS Key Management Service .

Autorisations de principal

Lorsque vous utilisez une clé gérée par le client pour chiffrer un package de déploiement .zip, seuls les principaux ayant accès à cette clé peuvent accéder au package de déploiement .zip. Par exemple, les principaux qui n'ont pas accès à la clé gérée par le client ne peuvent pas télécharger le package .zip à l'aide de l'URL S3 présignée incluse dans la réponse. GetFunction Une `AccessDeniedException` est renvoyée dans la section `Code` de la réponse.

Example AWS KMS AccessDeniedException

```
{
    "Code": {
        "RepositoryType": "S3",
        "Error": {
            "ErrorCode": "AccessDeniedException",
            "Message": "KMS access is denied. Check your KMS permissions. KMS
 Exception: AccessDeniedException KMS Message: User: arn:aws:sts::111122223333:assumed-
role/LambdaTestRole/session is not authorized to perform: kms:Decrypt on resource:
 arn:aws:kms:us-east-1:111122223333:key/key-id with an explicit deny in a resource-
based policy"
        },
        "SourceKMSKeyArn": "arn:aws:kms:us-east-1:111122223333:key/key-id"
    },
 ...
```

Pour plus d'informations sur les autorisations relatives aux AWS KMS clés, consultez Authentification et contrôle d'accès pour AWS KMS.

Utilisation d'une clé gérée par le client pour votre package de déploiement .zip

Utilisez les paramètres d'API suivants pour configurer les clés gérées par le client pour les packages de déploiement .zip :

- Source KMSKey Arn : chiffre le package de déploiement .zip source (le fichier que vous chargez).
- KMSKeyArn : chiffre les variables d'environnement et les instantanés SnapStart Lambda.

Lorsque `SourceKMSKeyArn` et `KMSKeyArn` sont tous deux spécifiés, Lambda utilise la clé `KMSKeyArn` pour chiffrer la version décompressée du package que Lambda utilise pour invoquer la fonction. Lorsque `SourceKMSKeyArn` est spécifié mais que `KMSKeyArn` ne l'est pas, Lambda utilise une Clé gérée par AWSpour chiffrer la version décompressée du package.

Lambda console

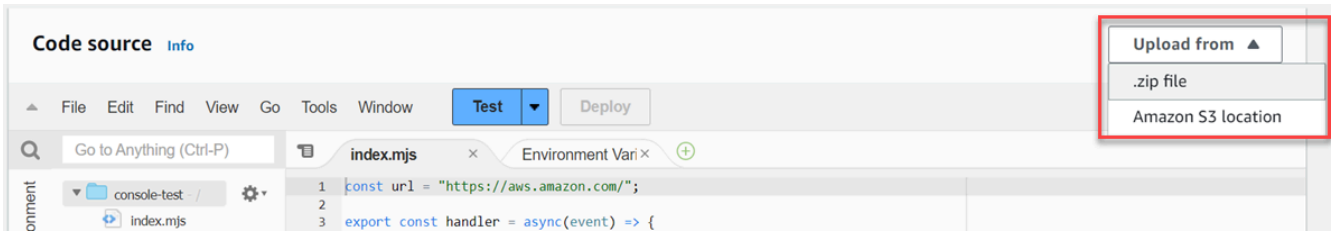
Pour ajouter le chiffrement par clé gérée par le client lors de la création d'une fonction

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez Créer une fonction.
3. Choisissez Author from scratch (Créer à partir de zéro) ou Container image (Image de conteneur).
4. Sous Basic information (Informations de base), procédez comme suit :
   a. Pour Function name (Nom de la fonction), saisissez le nom de la fonction.
   b. Pour Runtime, sélectionnez la version du langage à utiliser pour votre fonction.
5. Développez les paramètres avancés, puis sélectionnez Activer le chiffrement avec une clé gérée par AWS KMS le client.
6. Choisissez une clé gérée par le client.
7. Choisissez Créer une fonction.

Pour supprimer le chiffrement par clé gérée par le client ou pour utiliser une autre clé, vous devez télécharger à nouveau le package de déploiement .zip.

Pour ajouter le chiffrement par clé gérée par le client à une fonction existante

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom d'une fonction.
3. Dans le volet Source du code, choisissez Charger à partir de.
4. Choisissez un fichier .zip ou un emplacement Amazon S3.



5. Importez le fichier ou saisissez l'emplacement Amazon S3.
6. Choisissez Activer le chiffrement avec une clé gérée par le AWS KMS client.
7. Choisissez une clé gérée par le client.
8. Choisissez Enregistrer.

AWS CLI

Pour ajouter un chiffrement par clé gérée par le client lors de la création d'une fonction

Dans l'exemple create-function suivant :

- `--code`: Spécifie le chemin local vers le package de déploiement .zip (`ZipFile`) et la clé gérée par le client pour le chiffrer ()`SourceKMSKeyArn`.
- `--kms-key-arn` : spécifie la clé gérée par le client pour chiffrer les variables d'environnement et la version décompressée du package de déploiement.

```
aws lambda create-function \
  --function-name myFunction \
  --runtime nodejs24.x \
  --handler index.handler \
  --role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
  --code ZipFile=fileb://myFunction.zip,SourceKMSKeyArn=arn:aws:kms:us-
east-1:111122223333:key/key-id \
  --kms-key-arn arn:aws:kms:us-east-1:111122223333:key/key2-id
```

Dans l'exemple create-function suivant :

- `--code`: Spécifie l'emplacement du fichier .zip dans un compartiment Amazon S3 (`S3Bucket`,`S3Key`,`S3ObjectVersion`) et la clé gérée par le client pour le chiffrer (`SourceKMSKeyArn`).
- `--kms-key-arn` : spécifie la clé gérée par le client pour chiffrer les variables d'environnement et la version décompressée du package de déploiement.

```
aws lambda create-function \
  --function-name myFunction \
  --runtime nodejs24.x --handler index.handler \
  --role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
  --code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion,SourceKMSKeyArn=arn:aws:kms:us-
east-1:111122223333:key/key-id \
  --kms-key-arn arn:aws:kms:us-east-1:111122223333:key/key2-id
```

Pour ajouter un chiffrement par clé gérée par le client à une fonction existante

Dans l'update-function-codeexemple suivant :

- `--zip-file` : spécifie le chemin local du package de déploiement .zip.
- `--source-kms-key-arn` : spécifie la clé gérée par le client pour chiffrer la version compressée du package de déploiement. Lambda utilise une clé AWS détenue pour chiffrer le package décompressé pour les invocations de fonctions. Si vous souhaitez utiliser une clé gérée par le client pour chiffrer la version décompressée du package, exécutez la update-function-configurationcommande avec l'option. `--kms-key-arn`

```
aws lambda update-function-code \
  --function-name myFunction \
  --zip-file fileb://myFunction.zip \
  --source-kms-key-arn arn:aws:kms:us-east-1:111122223333:key/key-id
```

Dans l'update-function-codeexemple suivant :

- `--s3-bucket` : spécifie l'emplacement du fichier .zip dans un compartiment Amazon S3.
- `--s3-key` : spécifie la clé Amazon S3 du package de déploiement.
- `--s3-object-version` : pour les objets versionnés, la version de l'objet de package de déploiement à utiliser.
- `--source-kms-key-arn` : spécifie la clé gérée par le client pour chiffrer la version compressée du package de déploiement. Lambda utilise une clé AWS détenue pour chiffrer le package décompressé pour les invocations de fonctions. Si vous souhaitez utiliser une clé gérée par le client pour chiffrer la version décompressée du package, exécutez la update-function-configurationcommande avec l'option. `--kms-key-arn`

```
aws lambda update-function-code \
  --function-name myFunction \
  --s3-bucket amzn-s3-demo-bucket \
  --s3-key myFileName.zip \
  --s3-object-version myObject Version
  --source-kms-key-arn arn:aws:kms:us-east-1:111122223333:key/key-id
```

Pour supprimer le chiffrement par clé gérée par le client d'une fonction existante

Dans l'update-function-codeexemple suivant, `--zip-file` indique le chemin local vers le package de déploiement .zip. Lorsque vous exécutez cette commande sans `--source-kms-key-arn` option, Lambda utilise une clé AWS détenue pour chiffrer la version compressée du package de déploiement.

```
aws lambda update-function-code \
  --function-name myFunction \
  --zip-file fileb://myFunction.zip
```

Création d’une fonction Lambda à l’aide d’une image de conteneur

Le code de votre AWS Lambda fonction se compose de scripts ou de programmes compilés et de leurs dépendances. Pour déployer votre code de fonction vers Lambda, vous utilisez un package de déploiement. Lambda prend en charge deux types de packages de déploiement : les images conteneurs et les archives de fichiers .zip.

Il existe trois méthodes pour créer une image de conteneur pour une fonction Lambda :

- Utilisation d'une image AWS de base pour Lambda

  Les images de base AWS sont préchargées avec une exécution du langage, un client d’interface d’exécution pour gérer l’interaction entre Lambda et votre code de fonction, et un émulateur d’interface d’exécution pour les tests locaux.
- Utilisation d'une image de base AWS uniquement pour le système d'exploitation

  AWS Les images de base réservées au système d'exploitation contiennent une distribution Amazon Linux et l'émulateur d'interface d'exécution. Ces images sont couramment utilisées pour créer des images de conteneur pour les langages compilés, tels que Go et Rust, et pour une langue ou une version linguistique pour laquelle Lambda ne fournit pas d’image de base, comme Node.js 19. Vous pouvez également utiliser des images de base uniquement pour le système d'exploitation pour implémenter un environnement d'exécution personnalisé. Pour rendre l’image compatible avec Lambda, vous devez inclure un client d’interface d’exécution pour votre langage dans l’image.
- Utilisation d'une image non AWS basique

  Vous pouvez utiliser une autre image de base à partir d’un autre registre de conteneur, comme Alpine Linux ou Debian. Vous pouvez également utiliser une image personnalisée créée par votre organisation. Pour rendre l’image compatible avec Lambda, vous devez inclure un client d’interface d’exécution pour votre langage dans l’image.

> **Tip**
>
> Pour réduire le temps nécessaire à l’activation des fonctions du conteneur Lambda, consultez Utiliser des générations en plusieurs étapes (français non garanti) dans la documentation Docker. Pour créer des images de conteneur efficaces, suivez la section Bonnes pratiques pour l’écriture de Dockerfiles (français non garanti).

Pour créer une fonction Lambda à partir d'une image de conteneur, créez votre image localement et chargez-la dans un référentiel Amazon Elastic Container Registry (Amazon ECR). Si vous utilisez une image de conteneur fournie par un vendeur AWS Marketplace, vous devez d'abord cloner l'image dans votre référentiel Amazon ECR privé. Indiquez ensuite l'URI du référentiel lorsque vous créez la fonction. Le référentiel Amazon ECR doit être Région AWS identique à la fonction Lambda. Vous pouvez créer une fonction en utilisant une image d'un autre AWS compte, à condition que l'image se trouve dans la même région que la fonction Lambda. Pour de plus amples informations, veuillez consulter Autorisations entre comptes Amazon ECR.

> **Note**
>
> Lambda ne prend pas en charge les points de terminaison Amazon ECR FIPS pour les images de conteneurs. Si l'URI de votre dépôt contient `ecr-fips`, vous utilisez un point de terminaison FIPS. Exemple: `111122223333.dkr.ecr-fips.us-east-1.amazonaws.com`.

Cette page explique les types d'images de base et les exigences requises pour créer des images de conteneur compatibles avec Lambda.

> **Note**
>
> Vous ne pouvez pas modifier le type de package de déploiement (.zip ou image de conteneur) d'une fonction existante. Par exemple, vous ne pouvez pas convertir une fonction d'image de conteneur pour utiliser un fichier d'archive .zip à la place. Vous devez créer une nouvelle fonction.

Rubriques

- Exigences
- Utilisation d'une image AWS de base pour Lambda
- Utilisation d'une image de base AWS uniquement pour le système d'exploitation
- Utilisation d'une image non AWS basique
- Clients d'interface d'exécution
- Autorisations Amazon ECR
- Cycle de vie des fonctions

Exigences

Installez l'AWS CLI version 2 et la CLI Docker. En outre, notez les exigences suivantes :

- L'image de conteneur doit implémenter l'Utilisation de l'API de l'environnement d'exécution Lambda pour des environnements d'exécution personnalisés. Les clients d'interface de runtime open source AWS implémentent l'API. Vous pouvez ajouter un client d'interface de runtime à votre image de base préférée pour la rendre compatible avec Lambda.
- L'image de conteneur doit pouvoir s'exécuter sur un système de fichiers en lecture seule. Votre code de fonction peut accéder à un répertoire `/tmp` inscriptible entre 512 Mo et 10 240 Mo de stockage, par incréments de 1 Mo.
- L'utilisateur Lambda par défaut doit pouvoir lire tous les fichiers requis pour exécuter votre code de fonction. Lambda suit les bonnes pratiques de sécurité en définissant un utilisateur Linux par défaut avec des autorisations assortie d'un privilège minimum. Cela signifie que vous n'avez pas besoin de spécifier un UTILISATEUR dans votre Dockerfile. Vérifiez que votre code d'application ne repose pas sur des fichiers dont l'exécution n'est pas autorisée à d'autres utilisateurs Linux.
- Lambda prend uniquement en charge les images de conteneur basées sur Linux.
- Lambda fournit des images de base multi-architecture. Toutefois, l'image que vous créez pour la fonction ne doit avoir que l'une des architectures pour cible. Lambda ne prend pas en charge les fonctions qui utilisent les images de conteneur multi-architecture.

Utilisation d'une image AWS de base pour Lambda

Vous pouvez utiliser l'une des images de base AWS pour Lambda afin de créer l'image de conteneur pour votre code de fonction. Les images de base sont préchargées avec une exécution de langage et d'autres composants requis pour exécuter une image conteneur sur Lambda. Vous ajoutez votre code de fonction et vos dépendances à l'image de base, puis vous l'empaquetez en tant qu'image de conteneur.

AWS fournit régulièrement des mises à jour des images AWS de base pour Lambda. Si votre Dockerfile inclut le nom de l'image dans la propriété FROM, votre client Docker extrait la dernière version de l'image du référentiel Amazon ECR. Afin d'utiliser l'image de base mise à jour, vous devez reconstruire votre image de conteneur et mettre à jour le code de la fonction.

Les images de base de Node.js 20, Python 3.12, Java 21, .NET 8, Ruby 3.3 et versions ultérieures sont basées sur l'image de conteneur minimale Amazon Linux 2023. Les images de base antérieures utilisaient Amazon Linux 2. AL2Le 023 offre plusieurs avantages par rapport à Amazon Linux 2,

notamment un encombrement de déploiement réduit et des versions mises à jour de bibliothèques telles que `glibc`.

AL2Les images basées sur le format 023 utilisent `microdnf` (lien symbolique sous forme `dnf`) comme gestionnaire de packages au lieu de `yum`, qui est le gestionnaire de packages par défaut dans Amazon Linux 2. `microdnf` est une implémentation autonome de. `dnf` Pour obtenir la liste des packages inclus dans les images AL2 basées sur la version 023, reportez-vous aux colonnes Conteneur minimal de la section Comparaison des packages installés sur les images de conteneurs Amazon Linux 2023. Pour plus d'informations sur les différences entre AL2 023 et Amazon Linux 2, consultez Présentation du runtime Amazon Linux 2023 AWS Lambda sur le blog AWS Compute.

> **Note**
>
> Pour exécuter des images AL2 basées sur 023 localement, y compris avec AWS Serverless Application Model (AWS SAM), vous devez utiliser Docker version 20.10.10 ou ultérieure.

Pour créer une image de conteneur à l'aide d'une image de AWS base, choisissez les instructions correspondant à votre langue préférée :

- Node.js
- TypeScript(utilise une image de base Node.js)
- Python
- Java
- Go
- .NET
- Ruby

Utilisation d'une image de base AWS uniquement pour le système d'exploitation

AWS Les images de base réservées au système d'exploitation contiennent une distribution Amazon Linux et l'émulateur d'interface d'exécution. Ces images sont couramment utilisées pour créer des images de conteneur pour les langages compilés, tels que Go et Rust, et pour une langue ou une version linguistique pour laquelle Lambda ne fournit pas d’image de base, comme Node.js 19. Vous pouvez également utiliser des images de base uniquement pour le système d'exploitation pour

implémenter un environnement d'exécution personnalisé. Pour rendre l'image compatible avec Lambda, vous devez inclure un client d'interface d'exécution pour votre langage dans l'image.

Étiquettes	Environnement d'exécution	Système d'exploitation	Dockerfile	Obsolescence
al2023	Exécution uniquement basée sur le système d'exploitation	Amazon Linux	Dockerfile pour OS uniquement Runtime sur GitHub	30 juin 2029
al2	Exécution uniquement basée sur le système d'exploitation	Amazon Linux 2	Dockerfile pour OS uniquement Runtime sur GitHub	31 juillet 2026

Galerie publique d'Amazon Elastic Container Registry : gallery.ecr. aws/lambda/provided

Utilisation d'une image non AWS basique

Lambda prend en charge toute image conforme à l'un des formats de manifeste d'image suivants :

- Manifeste d'images Docker V2, schéma 2 (utilisé avec la version 1.10 de Docker et les versions les plus récentes)
- Spécifications OCI (Open Container initiative, soit initiative de conteneur ouvert) (v1.0.0 et versions ultérieures)

Lambda prend en charge une taille maximale d'image non compressée de 10 Go, comprenant toutes les couches.

> **Note**
>
> - Pour rendre l'image compatible avec Lambda, vous devez inclure un client d'interface d'exécution pour votre langage dans l'image.
> - Pour des performances optimales, maintenez la taille de votre manifeste d'image inférieure à 25 400 octets. Pour réduire la taille du manifeste d'image, réduisez le nombre de couches de votre image et réduisez le nombre d'annotations.

Clients d'interface d'exécution

Si vous utilisez une image de base uniquement pour le système d'exploitation ou une autre image de base, vous devez inclure un client d'interface d'exécution dans votre image. Le client de l'interface d'exécution doit étendre leUtilisation de l'API de l'environnement d'exécution Lambda pour des environnements d'exécution personnalisés, qui gère l'interaction entre Lambda et votre code de fonction. AWS fournit des clients d'interface d'exécution open source pour les langages suivants :

- Node.js
- Python
- Java
- .NET
- Go
- Ruby
- Rouille —

Si vous utilisez un langage qui ne possède pas de client d'interface d'exécution AWS fourni, vous devez créer le vôtre.

Autorisations Amazon ECR

Avant de créer une fonction Lambda à partir d'une image de conteneur, vous devez créer l'image localement et la charger dans un référentiel Amazon ECR. Lorsque vous créez la fonction, indiquez l'URI du référentiel Amazon ECR.

Vérifiez que les autorisations pour l'utilisateur ou le rôle qui crée la fonction contiennent `GetRepositoryPolicy` et `SetRepositoryPolicy`.

Par exemple, utilisez la console IAM pour créer un rôle avec la stratégie suivante :

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
        "ecr:SetRepositoryPolicy",
        "ecr:GetRepositoryPolicy"
      ],
      "Resource": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world"
    }
  ]
}
```

Politiques de référentiel Amazon ECR

Pour qu'une fonction dans le même compte accède à l'image conteneur dans Amazon ECR, vous pouvez ajouter des autorisations `ecr:BatchGetImage` et `ecr:GetDownloadUrlForLayer` à votre politique de référentiel Amazon ECR. L'exemple suivant présente la stratégie minimum :

```
{
        "Sid": "LambdaECRImageRetrievalPolicy",
        "Effect": "Allow",
        "Principal": {
          "Service": "lambda.amazonaws.com"
        },
        "Action": [
          "ecr:BatchGetImage",
          "ecr:GetDownloadUrlForLayer"
        ]
    }
```

Pour plus d'informations sur les autorisations des référentiels Amazon ECR, consultez Politiques des référentiels privés dans le Guide de l'utilisateur Amazon Elastic Container Registry.

Si le référentiel Amazon ECR n'inclut pas ces autorisations, Lambda tente de les ajouter automatiquement. Lambda ne peut ajouter ces autorisations que si le principal appelant Lambda dispose des autorisations `ecr:getRepositoryPolicy` et `ecr:setRepositoryPolicy`.

Pour afficher ou modifier les autorisations de votre référentiel Amazon ECR, suivez les instructions de la section Définition d'une déclaration de politique de référentiel privé dans le Guide de l'utilisateur Amazon Elastic Container Registry.

Autorisations entre comptes Amazon ECR

Un autre compte dans la même région peut créer une fonction qui utilise une image conteneur appartenant à votre compte. Dans l'exemple suivant, votre politique d'autorisations de référentiel Amazon ECR a besoin des instructions suivantes pour accorder l'accès au numéro de compte 123456789012.

- CrossAccountPermission— Permet au compte 123456789012 de créer et de mettre à jour des fonctions Lambda qui utilisent des images provenant de ce référentiel ECR.
- Lambda — ECRImage CrossAccountRetrievalPolicy Lambda finira par définir l'état d'une fonction sur inactif si elle n'est pas invoquée pendant une période prolongée. Cette instruction est requise pour que Lambda puisse récupérer l'image conteneur à des fins d'optimisation et de mise en cache pour le compte de la fonction détenue par 123456789012.

Example – Ajouter une autorisation entre comptes à votre référentiel

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "CrossAccountPermission",
      "Effect": "Allow",
      "Action": [
        "ecr:BatchGetImage",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Principal": {
        "AWS": "arn:aws:iam::123456789012:root"
      },
```

```
      "Resource": "arn:aws:ecr:us-east-1:123456789012:repository/example-lambda-
repository"
    },
    {
      "Sid": "LambdaECRImageCrossAccountRetrievalPolicy",
      "Effect": "Allow",
      "Action": [
        "ecr:BatchGetImage",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Condition": {
        "ArnLike": {
          "aws:sourceARN": "arn:aws:lambda:us-east-1:123456789012:function:*"
        }
      },
      "Resource": "arn:aws:ecr:us-east-1:123456789012:repository/example-lambda-
repository"
    }
  ]
}
```

Pour donner accès à plusieurs comptes, vous ajoutez le compte IDs à la liste Principal de la `CrossAccountPermission` politique et à la liste d'évaluation des conditions dans le`LambdaECRImageCrossAccountRetrievalPolicy`.

Si vous travaillez avec plusieurs comptes au AWS sein d'une organisation, nous vous recommandons d'énumérer chaque ID de compte dans la politique d'autorisation ECR. Cette approche est conforme aux meilleures pratiques de AWS sécurité qui consistent à définir des autorisations restreintes dans les politiques IAM.

Outre les autorisations Lambda, l'utilisateur ou le rôle qui crée la fonction doit également disposer des autorisations `BatchGetImage` et `GetDownloadUrlForLayer`.

Cycle de vie des fonctions

Après avoir chargé une nouvelle image de conteneur ou une image de conteneur mise à jour, Lambda optimise l'image avant que la fonction ne puisse traiter les appels. Le processus d'optimisation peut prendre quelques secondes. La fonction reste dans l'état `Pending` jusqu'à ce

que le processus se termine, l'état passant alors à `Active`. Vous ne pouvez pas invoquer la fonction jusqu'à ce qu'elle atteigne l'état `Active`.

Si une fonction n'est pas appelée pendant plusieurs semaines, Lambda récupère sa version optimisée, et la fonction passe à l'état `Inactive`. Pour réactiver la fonction, vous devez l'appeler. Lambda rejette la première invocation et la fonction passe à l'état `Pending` jusqu'à ce que Lambda réoptimise l'image. La fonction revient ensuite à l'état `Active`.

Lambda récupère périodiquement l'image de conteneur associée à partir du référentiel Amazon ECR. Si l'image de conteneur correspondante n'existe plus sur Amazon ECR ou si les autorisations ont été révoquées, la fonction passera à l'état `Failed`, et Lambda renverra un message d'échec à tout appel de la fonction.

Vous pouvez utiliser l'API Lambda pour obtenir des informations sur l'état d'une fonction. Pour de plus amples informations, veuillez consulter États de la fonction Lambda.

Configuration de la mémoire d'une fonction Lambda

Lambda alloue de la puissance d'UC en fonction de la quantité de mémoire configurée. La mémoire est la quantité de mémoire disponible pour une fonction Lambda lors de l'exécution. Vous pouvez augmenter ou réduire la mémoire et la puissance d'UC allouées à votre fonction à l'aide du paramètre Mémoire. Vous pouvez configurer la mémoire comprise entre 128 et 10 240 Mo, par incréments de 1 Mo. A 1 769 Mo, une fonction dispose de l'équivalent d'1 vCPU (un vCPU-seconde de crédits par seconde).

Cette page explique comment et quand mettre à jour le paramètre de mémoire pour une fonction Lambda.

Sections

- Évaluation du paramètre de mémoire approprié pour une fonction Lambda
- Configuration de la mémoire d'une fonction (console)
- Configuration de la mémoire d'une fonction (AWS CLI)
- Configuration de la mémoire d'une fonction (AWS SAM)
- Acceptation des recommandations relatives à la mémoire d'une fonction (console)

Évaluation du paramètre de mémoire approprié pour une fonction Lambda

La mémoire est le principal instrument de contrôle de l'exécution d'une fonction. Le réglage par défaut, 128 Mo, est le plus bas possible. Nous vous recommandons de n'utiliser que 128 Mo pour les fonctions Lambda simples, telles que celles qui transforment et acheminent des événements vers d'autres services AWS . Une allocation de mémoire plus élevée peut améliorer les performances pour les fonctions qui utilisent des bibliothèques importées, des couches Lambda, Amazon Simple Storage Service (Amazon S3) ou Amazon Elastic File System (Amazon EFS). L'ajout de mémoire augmente proportionnellement la quantité de processeur, augmentant ainsi la puissance de calcul globale disponible. Si une fonction est liée au processeur, au réseau ou à la mémoire, l'augmentation du paramètre de mémoire peut améliorer considérablement ses performances.

Pour trouver la bonne configuration de mémoire, surveillez vos fonctions avec Amazon CloudWatch et définissez des alarmes si la consommation de mémoire approche les valeurs maximales configurées. Cela peut aider à identifier les fonctions liées à la mémoire. Pour les fonctions liées au processeur et aux E/S, le suivi de la durée apporte souvent des informations intéressantes. Dans ces cas, l'augmentation de la mémoire peut aider à résoudre les goulets d'étranglement de calcul ou de réseau.

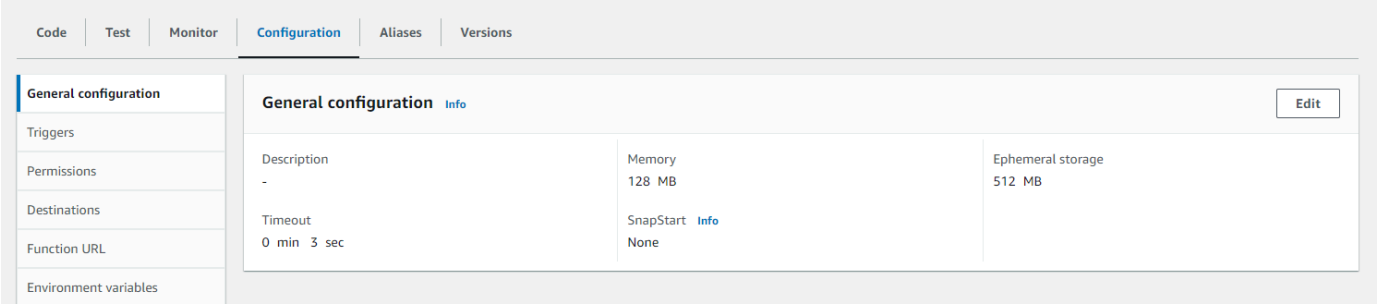
Vous pouvez également envisager d'utiliser l'outil open source AWS Lambda Power Tuning. Cet outil permet AWS Step Functions d'exécuter plusieurs versions simultanées d'une fonction Lambda avec différentes allocations de mémoire et de mesurer les performances. La fonction de saisie s'exécute dans votre AWS compte et effectue des appels HTTP en direct et une interaction avec le SDK, afin de mesurer les performances probables dans un scénario de production en direct. Vous pouvez également implémenter un CI/CD processus permettant d'utiliser cet outil pour mesurer automatiquement les performances des nouvelles fonctions que vous déployez.

Configuration de la mémoire d'une fonction (console)

Vous pouvez configurer la mémoire de votre fonction dans la console Lambda.

Pour mettre à jour la mémoire d'une fonction

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Choisissez Configuration, puis Configuration générale.



4. Sous Configuration générale, choisissez Modifier.
5. Pour Mémoire, définissez une valeur comprise entre 128 et 10 240 Mo.
6. Choisissez Enregistrer.

Configuration de la mémoire d'une fonction (AWS CLI)

Vous pouvez utiliser la update-function-configurationcommande pour configurer la mémoire de votre fonction.

Example

```
aws lambda update-function-configuration \
  --function-name my-function \
```

```
  --memory-size 1024
```

Configuration de la mémoire d'une fonction (AWS SAM)

Vous pouvez utiliser AWS Serverless Application Model pour configurer la mémoire de votre fonction. Mettez à jour la MemorySizepropriété dans votre `template.yaml` fichier, puis exécutez sam deploy.

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 1024
      # Other function properties...
```

Acceptation des recommandations relatives à la mémoire d'une fonction (console)

Si vous disposez d'autorisations d'administrateur dans Gestion des identités et des accès AWS (IAM), vous pouvez choisir de recevoir des recommandations de configuration de la mémoire des fonctions Lambda de la part de. Optimiseur de calcul AWS Pour obtenir des instructions sur la manière de choisir de recevoir des recommandations relatives à la mémoire pour votre compte ou votre organisation, consultez Inscription à votre compte dans le Guide de l'utilisateur Optimiseur de calcul AWS .

> **Note**
>
> Compute Optimizer prend en charge uniquement les fonctions qui utilisent l'architecture x86_64.

Si vous avez choisi cette option et si votre fonction Lambda répond aux exigences de Compute Optimizer, vous pouvez afficher et accepter les recommandations de Compute Optimizer relatives à la mémoire de la fonction dans la console Lambda dans Configuration générale.

Configuration d’un stockage éphémère pour les fonctions Lambda

Lambda fournit un stockage éphémère pour les fonctions du répertoire `/tmp`. Ce stockage est temporaire et propre à chaque environnement d’exécution. Vous pouvez contrôler la quantité de stockage éphémère allouée à votre fonction à l’aide du paramètre Stockage éphémère. Vous pouvez configurer le stockage éphémère comprise entre 512 et 10 240 Mo, par incréments de 1 Mo. Toutes les données stockées dans `/tmp` sont chiffrées au repos à l’aide d’une clé gérée par AWS.

Cette page décrit les cas d’utilisation courants et explique comment mettre à jour le stockage éphémère pour une fonction Lambda.

Sections

- Cas d’utilisation courants pour un stockage éphémère augmenté
- Configuration du magasin éphémère (console)
- Configuration du stockage éphémère (AWS CLI)
- Configuration du stockage éphémère (AWS SAM)

Cas d’utilisation courants pour un stockage éphémère augmenté

Voici plusieurs cas d’utilisation courants qui bénéficient d’un stockage éphémère augmenté :

- Extract-transform-load Tâches (ETL) : augmentez le stockage éphémère lorsque votre code effectue des calculs intermédiaires ou télécharge d'autres ressources pour terminer le traitement. Un espace temporaire plus important permet d’exécuter des tâches ETL plus complexes dans des fonctions Lambda.
- Inférence de machine learning (ML) : de nombreuses tâches d’inférence reposent sur des fichiers de données de référence volumineux, notamment des bibliothèques et des modèles. Avec un stockage éphémère plus important, vous pouvez télécharger des modèles plus grands depuis Amazon Simple Storage Service (Amazon S3) vers `/tmp` et les utiliser pour votre traitement.
- Traitement des données : pour les charges de travail qui téléchargent des objets depuis Amazon S3 en réponse à des événements S3, un espace `/tmp` plus important permet de gérer des objets plus volumineux sans recourir au traitement en mémoire. Les charges de travail qui créent PDFs ou traitent des médias bénéficient également d'un stockage plus éphémère.
- Traitement graphique : le traitement d’image est un cas d’utilisation courant pour les applications basées sur Lambda. Pour les charges de travail qui traitent de gros fichiers TIFF ou des images

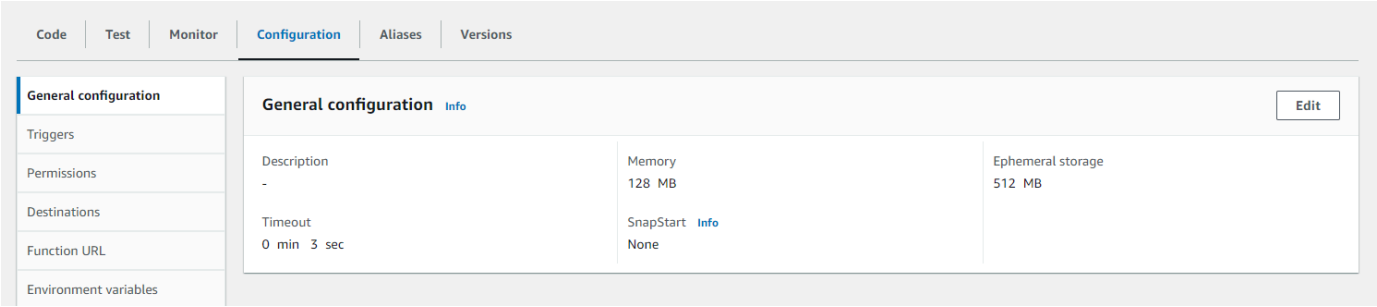
satellites, un stockage éphémère plus important facilite l'utilisation des bibliothèques et l'exécution des calculs dans Lambda.

Configuration du magasin éphémère (console)

Vous pouvez configurer le stockage éphémère dans la console Lambda.

Pour modifier le stockage éphémère d'une fonction

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Choisissez Configuration, puis Configuration générale.



4. Sous Configuration générale, choisissez Modifier.
5. Pour Stockage éphémère, définissez une valeur comprise entre 512 et 10 240 Mo, par incréments de 1 Mo.
6. Choisissez Enregistrer.

Configuration du stockage éphémère (AWS CLI)

Vous pouvez utiliser la update-function-configurationcommande pour configurer le stockage éphémère.

Example

```
aws lambda update-function-configuration \
  --function-name my-function \
  --ephemeral-storage '{"Size": 1024}'
```

Configuration du stockage éphémère (AWS SAM)

Vous pouvez utiliser AWS Serverless Application Model pour configurer le stockage éphémère pour votre fonction. Mettez à jour la EphemeralStoragepropriété dans votre `template.yaml` fichier, puis exécutez sam deploy.

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 120
      Handler: index.handler
      Runtime: nodejs22.x
      Architectures:
        - x86_64
      EphemeralStorage:
        Size: 10240
      # Other function properties...
```

Sélection et configuration d’une architecture de jeu d’instructions pour votre fonction Lambda

L'architecture de l'ensemble des instructions d'une fonction Lambda détermine le type de processeur d'ordinateur que Lambda utilise pour exécuter la fonction. Lambda permet de choisir parmi plusieurs architectures de l'ensemble des instructions :

- arm64 — Architecture ARM 64 bits, pour le processeur AWS Graviton2.
- x86_64 : architecture 64 bits x86, pour les processeurs x86.

> **Note**
>
> L'architecture arm64 est disponible dans la plupart des Régions AWS cas. Pour plus d’informations, consultez Tarification d’AWS Lambda. Dans le tableau des prix de la mémoire, choisissez l'onglet Arm Price, puis ouvrez la liste déroulante des régions pour voir qui Régions AWS prend en charge arm64 avec Lambda.
> Pour un exemple de création d'une fonction avec l'architecture arm64, voir AWS Lambda Fonctions alimentées par le processeur AWS Graviton2.

Rubriques

- Avantages liés à l'utilisation de l'architecture arm64
- Exigences pour la migration vers l'architecture arm64
- Compatibilité des codes de fonction avec l'architecture arm64
- Comment migrer vers l'architecture arm64
- Configuration de l'architecture de l'ensemble des instructions

Avantages liés à l'utilisation de l'architecture arm64

Les fonctions Lambda qui utilisent l'architecture arm64 (processeur AWS Graviton2) peuvent atteindre un prix et des performances nettement supérieurs à ceux des fonctions équivalentes exécutées sur l'architecture x86_64. Envisagez d'utiliser arm64 pour les applications de calcul intensif telles que le calcul haute performance, le codage vidéo et les applications de simulation.

Le processeur Graviton2 utilise le cœur Neoverse N1 et prend en charge Armv8.2 (y compris les extensions CRC et cryptographiques), ainsi que plusieurs autres extensions architecturales.

Graviton2 réduit le temps de lecture de la mémoire en fournissant un cache L2 plus important par vCPU, ce qui permet d'obtenir une meilleure performance de latence des backends Web et mobiles, des microservices et des systèmes de traitement de données. Graviton2 offre également des performances de chiffrement améliorées et prend en charge les ensembles des instructions qui améliorent la latence de l'inférence Machine Learning basée sur CPU.

Pour plus d'informations sur AWS Graviton2, consultez la section Processeur AWS Graviton.

Exigences pour la migration vers l'architecture arm64

Lorsque vous sélectionnez une fonction Lambda à migrer vers l'architecture arm64, pour que la migration soit fluide, assurez-vous que la fonction répond aux exigences suivantes :

- Le package de déploiement ne contient que des composants open source et le code source que vous contrôlez, ce qui vous permet d'effectuer les mises à jour nécessaires à la migration.
- Si le code de la fonction inclut des dépendances tierces, chaque bibliothèque ou package fournit une version arm64.

Compatibilité des codes de fonction avec l'architecture arm64

Le code de la fonction Lambda doit être compatible avec l'architecture de l'ensemble des instructions de la fonction. Avant de migrer une fonction vers l'architecture arm64, veuillez noter les points suivants concernant le code de fonction actuel :

- Si vous avez ajouté le code de la fonction à l'aide de l'éditeur de code intégré, le code s'exécute probablement sur l'une des architectures sans modification.
- Si vous avez chargé le code de la fonction, vous devez charger un nouveau code compatible avec l'architecture cible.
- Si la fonction utilise des couches, vous devez vérifier chaque couche afin de vous assurer qu'elle est compatible avec la nouvelle architecture. Si une couche n'est pas compatible, modifiez la fonction, afin de remplacer la version actuelle de la couche par une version de couche compatible.
- Si la fonction utilise des extensions Lambda, vous devez vérifier chaque extension, afin de vous assurer de sa compatibilité avec la nouvelle architecture.

- Si votre fonction utilise un type de package de déploiement de l'image de conteneur, vous devez créer une nouvelle image de conteneur compatible avec l'architecture de la fonction.

Comment migrer vers l'architecture arm64

Pour migrer une fonction Lambda vers l'architecture arm64, nous recommandons de procéder comme suit :

1. Créez la liste des dépendances de l'application ou de la charge de travail. Les dépendances courantes comprennent :
   - Toutes les bibliothèques et les packages utilisés par la fonction.
   - Les outils que vous utilisez pour créer, déployer et tester la fonction, tels que les compilateurs, les suites de tests, les pipelines d'intégration continue et de livraison continue (CI/CD), les outils d'approvisionnement et les scripts.
   - Les extensions Lambda et les outils tiers que vous utilisez pour contrôler la fonction en production.
2. Pour chacune des dépendances, vérifiez la version, puis vérifiez si les versions arm64 sont disponibles.
3. Créez un environnement pour migrer l'application.
4. Amorcez l'application.
5. Testez et déboguez l'application.
6. Testez la performance de la fonction arm64. Comparez la performance avec celle de la version x86_64.
7. Mettez à jour le pipeline d'infrastructure pour prendre en charge les fonctions Lambda arm64.
8. Établissez le déploiement vers la production.

   Par exemple, utilisez la configuration du routage d'alias pour fractionner le trafic entre les versions x86 et arm64 de la fonction et comparez les performances et la latence.

Pour plus d'informations sur la création d'un environnement de code pour l'architecture arm64, y compris des informations spécifiques au langage pour Java, Go, .NET et Python, consultez le référentiel Getting started with AWS Graviton. GitHub

Configuration de l'architecture de l'ensemble des instructions

Vous pouvez configurer l'architecture du jeu d'instructions pour les fonctions Lambda nouvelles et existantes à l'aide de la console Lambda, AWS SDKs, AWS Command Line Interface ()AWS CLI ou. CloudFormation Procédez comme suit pour modifier l'architecture de l'ensemble des instructions d'une fonction Lambda existante à partir de la console.

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom de la fonction pour laquelle vous voulez configurer l'architecture de l'ensemble des instructions.
3. Dans l'onglet principal Code, pour la section Paramètres d'exécution, choisissez Modifier.
4. Sous Architecture, choisissez l'architecture de l'ensemble des instructions que votre fonction doit utiliser.
5. Choisissez Enregistrer.

Configuration du délai d'expiration d'une fonction Lambda

Lambda exécute votre code pendant une durée définie avant l'expiration du délai. Le délai d'expiration est la durée maximale, en secondes, pendant laquelle une fonction Lambda peut être exécutée. La valeur par défaut de ce paramètre est de 3 secondes, mais vous pouvez l'ajuster par incréments d'1 seconde jusqu'à une valeur maximale de 900 secondes (15 minutes).

Cette page explique comment et quand mettre à jour le paramètre de délai d'expiration pour une fonction Lambda.

Sections

- Évaluation de la valeur de délai d'expiration appropriée pour une fonction Lambda
- Configuration du délai d'expiration (console)
- Configuration du délai d'expiration (AWS CLI)
- Configuration du délai d'expiration (AWS SAM)

Évaluation de la valeur de délai d'expiration appropriée pour une fonction Lambda

Si la valeur du délai d'expiration est proche de la durée moyenne d'une fonction, il y a un risque plus élevé que la fonction s'arrête de manière inattendue. La durée d'exécution d'une fonction peut varier en fonction de la quantité de données transférées et traitées, ainsi que de la latence des services avec lesquels la fonction interagit. Parmi les causes courantes de délai d'expiration, citons les suivantes :

- Les téléchargements à partir d'Amazon Simple Storage Service (Amazon S3) sont plus importants ou prennent plus de temps que d'habitude.
- Une fonction adresse une requête à un autre service, lequel met plus de temps à répondre.
- Les paramètres fournis à une fonction exigent une plus grande complexité de calcul dans la fonction, ce qui allonge la durée de l'invocation.

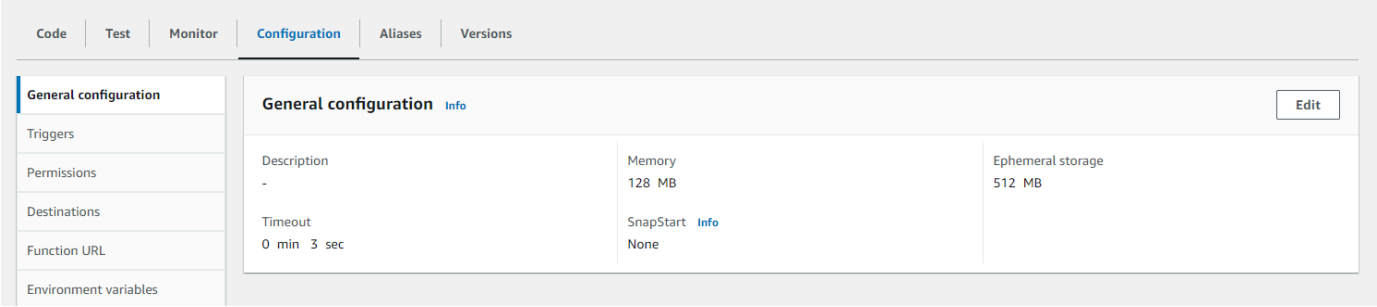
Lors des tests de votre application, veillez à ce que ceux-ci reflètent fidèlement la taille et la quantité des données, et à ce que les valeurs des paramètres soient réalistes. Les tests utilisent souvent de petits échantillons pour des raisons de commodité. Cependant, les jeux de données utilisés devraient se rapprocher des limites supérieures de ce que l'on peut raisonnablement attendre de votre charge de travail.

Configuration du délai d'expiration (console)

Vous pouvez configurer le délai d'expiration d'une fonction dans la console Lambda.

Pour modifier le délai d'expiration d'une fonction

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Choisissez Configuration, puis Configuration générale.



4. Sous Configuration générale, choisissez Modifier.
5. Pour Expiration, définissez une valeur comprise entre 1 et 900 secondes (15 minutes).
6. Choisissez Enregistrer.

Configuration du délai d'expiration (AWS CLI)

Vous pouvez utiliser la update-function-configurationcommande pour configurer la valeur du délai d'expiration, en secondes. L'exemple de commande suivant augmente le délai d'expiration de la fonction à 120 secondes (2 minutes).

Example

```
aws lambda update-function-configuration \
  --function-name my-function \
  --timeout 120
```

Configuration du délai d'expiration (AWS SAM)

Vous pouvez utiliser AWS Serverless Application Model pour configurer la valeur du délai d'expiration de votre fonction. Mettez à jour la propriété Timeout dans votre fichier `template.yaml`, puis exécutez sam deploy.

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 120
      # Other function properties...
```

Configurer les fonctions durables de Lambda

Pour permettre une exécution durable de votre fonction Lambda, vous devez configurer des paramètres spécifiques qui contrôlent la durée d'exécution de votre fonction, la durée de conservation des données d'état et les autorisations requises.

Permettre une exécution durable

Pour permettre une exécution durable de votre fonction Lambda, configurez-la `DurableConfig` dans la configuration de votre fonction. Ce paramètre contrôle le délai d'exécution, la conservation des états et le comportement de gestion des versions.

AWS CLI

```
aws lambda update-function-configuration \
  --function-name my-durable-function \
  --durable-config '{
    "ExecutionTimeout": 3600,
    "RetentionPeriodInDays": 30,
    "AllowInvokeLatest": true
  }'
```

CloudFormation

```
Resources:
  MyDurableFunction:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: my-durable-function
      Runtime: nodejs18.x
      Handler: index.handler
      Code:
        ZipFile: |
          // Your durable function code
      DurableConfig:
        ExecutionTimeout: 3600
        RetentionPeriodInDays: 30
        AllowInvokeLatest: true
```

Paramètres de configuration :

- `ExecutionTimeout`- Temps d'exécution maximal en secondes (jusqu'à 31 536 000 pour un an)
- `RetentionPeriodInDays`- Combien de temps faut-il conserver l'état et l'historique d'exécution (1 à 365 jours)
- `AllowInvokeLatest`- Autoriser ou non l'invocation de la version $LATEST pour une exécution durable

Autorisations IAM pour des fonctions durables

Les fonctions durables nécessitent des autorisations IAM supplémentaires au-delà des rôles d'exécution Lambda standard. Le rôle d'exécution de votre fonction doit inclure des autorisations pour la gestion des états et l'exécution durable APIs.

Autorisations minimales requises :

```
{
  "Version": "2012-10-17",
  "Statement": [
```

```
    {
      "Effect": "Allow",
      "Action": [
        "lambda:InvokeFunction",
        "lambda:GetFunction",
        "lambda:ManageDurableState",
        "lambda:GetDurableExecution",
        "lambda:ListDurableExecutions"
      ],
      "Resource": "arn:aws:lambda:*:*:function:*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": "arn:aws:logs:*:*:*"
    }
  ]
}
```

CloudFormation exemple de rôle d'exécution :

```
DurableFunctionRole:
  Type: AWS::IAM::Role
  Properties:
    AssumeRolePolicyDocument:
      Version: '2012-10-17'
      Statement:
        - Effect: Allow
          Principal:
            Service: lambda.amazonaws.com
          Action: sts:AssumeRole
    ManagedPolicyArns:
      - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
    Policies:
      - PolicyName: DurableFunctionPolicy
        PolicyDocument:
          Version: '2012-10-17'
```

```
          Statement:
            - Effect: Allow
              Action:
                - lambda:ManageDurableState
                - lambda:GetDurableExecution
                - lambda:ListDurableExecutions
              Resource: '*'
```

Bonnes pratiques en matière de configuration

Suivez les meilleures pratiques suivantes lors de la configuration de fonctions durables pour une utilisation en production :

- Définissez des délais d'exécution appropriés - Configurez `ExecutionTimeout` en fonction de la durée maximale prévue de votre flux de travail. Ne fixez pas de délais d'attente inutilement longs, car ils affectent les coûts et l'allocation des ressources.
- Équilibrez la rétention avec les coûts de stockage : définissez les `RetentionPeriodInDays` paramètres en fonction de vos exigences en matière de débogage et d'audit. Des périodes de conservation plus longues augmentent les coûts de stockage.
- Utiliser le versionnement en production : configurez-le sur `false` dans `AllowInvokeLatest` les environnements de production et utilisez des versions de fonctions ou des alias spécifiques pour des exécutions durables.
- Taille de l'état du moniteur : les objets à état volumineux augmentent les coûts de stockage et peuvent avoir un impact sur les performances. Maintenez l'état minimal et utilisez un stockage externe pour les données volumineuses.
- Configurer une journalisation appropriée - Activez la journalisation détaillée pour résoudre les problèmes liés aux flux de travail de longue durée, tout en tenant compte du volume des journaux et des coûts.

Exemple de configuration de production :

```
{
  "ExecutionTimeout": 86400,
  "RetentionPeriodInDays": 7,
  "AllowInvokeLatest": false
}
```

Utilisation des variables d’environnement Lambda

Vous pouvez utiliser des variables d’environnement pour ajuster le comportement de votre fonction sans mettre à jour le code. Une variable d’environnement est une paire de chaînes stockées dans la configuration spécifique à la version d’une fonction. L’environnement d’exécution de Lambda met les variables d’environnement à la disposition de votre code et définit les variables d’environnement supplémentaires qui contiennent des informations sur la fonction et la demande d’invocation.

> **Note**
>
> Pour renforcer la sécurité, nous vous recommandons d'utiliser à la AWS Secrets Manager place des variables d'environnement pour stocker les informations d'identification de la base de données et d'autres informations sensibles telles que les clés d'API ou les jetons d'autorisation. Pour de plus amples informations, veuillez consulter Utiliser des secrets Secrets Manager dans des fonctions Lambda.

Les variables d’environnement ne sont pas évaluées avant l’invocation de la fonction. Toute valeur que vous définissez est considérée comme une chaîne littérale et non développée. Effectuez l’évaluation de la variable dans le code de votre fonction.

Création de variables d’environnement Lambda

Vous pouvez configurer les variables d'environnement dans Lambda à l'aide de la console Lambda, du AWS Command Line Interface (AWS CLI), AWS Serverless Application Model (AWS SAM) ou d'un SDK. AWS

Console

Vous définissez des variables d’environnement sur la version non publiée de votre fonction. Lorsque vous publiez une version, les variables d’environnement sont verrouillées pour cette version ainsi que d’autres paramètres de configuration spécifiques à la version.

Vous créez une variable d’environnement pour votre fonction en définissant une clé et une valeur. Votre fonction utilise le nom de la clé pour récupérer la valeur de la variable d’environnement.

Pour définir des variables d’environnement dans la console Lambda

1. Ouvrez la page Functions (Fonctions) de la console Lambda.

2. Choisissez une fonction.
3. Choisissez Configuration, puis Variables d'environnement.
4. Sous Variables d'environnement, choisissez Modifier.
5. Choisissez Ajouter une variable d'environnement.
6. Entrez une clé et une valeur.

   Exigences

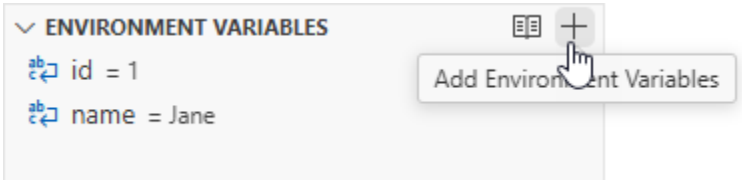
   - Les clés commencent par une lettre et comportent au moins deux caractères.
   - Les clés contiennent uniquement des lettres, des chiffres et le caractère de soulignement (_).
   - Les clés ne sont pas réservées par Lambda.
   - La taille totale de toutes les variables d'environnement ne dépasse pas 4 Ko.
7. Choisissez Enregistrer.

Pour générer une liste de variables d'environnement dans l'éditeur de code de la console

Vous pouvez générer une liste de variables d'environnement dans l'éditeur de code Lambda. Il s'agit d'un moyen rapide de référencer vos variables d'environnement pendant que vous codez.

1. Cliquez sur l'onglet Code.
2. Faites défiler l'écran jusqu'à la section VARIABLES D'ENVIRONNEMENT de l'éditeur de code. Les variables d'environnement existantes sont répertoriées ici :

3. Pour créer de nouvelles variables d'environnement, choisissez le signe plus (+) :



Les variables d'environnement restent chiffrées lorsqu'elles sont répertoriées dans l'éditeur de code de la console. Si vous avez activé les assistants de chiffrement pour le chiffrement en transit, ces paramètres restent inchangés. Pour de plus amples informations, veuillez consulter Sécurisation de variables d'environnement Lambda.

La liste des variables d'environnement est en lecture seule et n'est disponible que sur la console Lambda. Ce fichier n'est pas inclus lorsque vous téléchargez l'archive .zip de la fonction et vous ne pouvez pas ajouter de variables d'environnement en chargeant ce fichier.

AWS CLI

L'exemple suivant définit deux variables d'environnement sur une fonction nommée `my-function`.

```
aws lambda update-function-configuration \
  --function-name my-function \
  --environment "Variables={BUCKET=amzn-s3-demo-bucket,KEY=file.txt}"
```

Lorsque vous appliquez des variables d'environnement avec la commande `update-function-configuration`, l'ensemble du contenu de la structure `Variables` est remplacé. Pour conserver les variables d'environnement existantes lorsque vous en ajoutez une nouvelle, incluez toutes les valeurs existantes dans votre demande.

Pour obtenir la configuration actuelle, utilisez la commande `get-function-configuration`.

```
aws lambda get-function-configuration \
  --function-name my-function
```

Vous devriez voir la sortie suivante :

```
{
    "FunctionName": "my-function",
    "FunctionArn": "arn:aws:lambda:us-east-2:111122223333:function:my-function",
    "Runtime": "nodejs24.x",
    "Role": "arn:aws:iam::111122223333:role/lambda-role",
    "Environment": {
        "Variables": {
            "BUCKET": "amzn-s3-demo-bucket",
            "KEY": "file.txt"
        }
    },
    "RevisionId": "0894d3c1-2a3d-4d48-bf7f-abade99f3c15",
```

```
    ...
}
```

Vous pouvez transmettre l'identifiant de révision depuis la sortie de `get-function-configuration` en tant que paramètre à `update-function-configuration`. Cela garantit que les valeurs ne changent pas entre le moment où vous lisez la configuration et celui où vous la mettez à jour.

Pour configurer la clé de chiffrement d'une fonction, définissez l'option `KMSKeyARN`.

```
aws lambda update-function-configuration \
  --function-name my-function \
  --kms-key-arn arn:aws:kms:us-east-2:111122223333:key/055efbb4-xmpl-4336-
ba9c-538c7d31f599
```

AWS SAM

Vous pouvez utiliser AWS Serverless Application Model pour configurer les variables d'environnement de votre fonction. Mettez à jour les propriétés Environnement et Variables dans votre fichier `template.yaml`, puis exécutez sam deploy.

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 120
      Handler: index.handler
      Runtime: nodejs24.x
      Architectures:
        - x86_64
      EphemeralStorage:
        Size: 10240
      Environment:
        Variables:
```

```
        BUCKET: amzn-s3-demo-bucket
        KEY: file.txt
    # Other function properties...
```

AWS SDKs

Pour gérer les variables d'environnement à l'aide d'un AWS SDK, utilisez les opérations d'API suivantes.

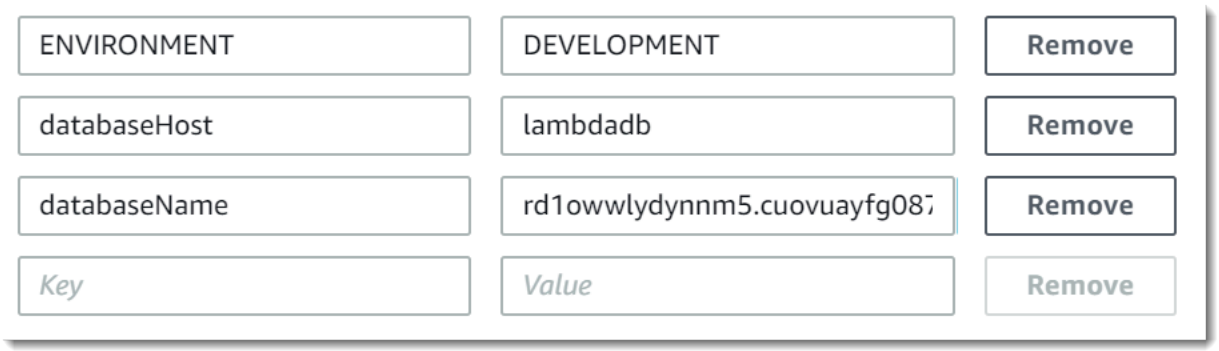
- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

Pour en savoir plus, référez-vous à la documentation du kit SDK AWS correspondant à votre langage de programmation préféré.

Exemple de scénario pour les variables d'environnement

Vous pouvez utiliser des variables d'environnement pour personnaliser le comportement des fonctions dans vos environnements de test et de production. Vous pouvez par exemple créer deux fonctions avec le même code mais une configuration différente. Une fonction se connecte à une base de données de test et l'autre à une base de données de production. Dans ce cas, vous utilisez des variables d'environnement pour transmettre à la fonction le nom d'hôte, ainsi que d'autres détails de connexion pour la base de données.

L'exemple suivant montre comment définir l'hôte de la base de données et le nom de base de données en tant que variables d'environnement.



Si vous souhaitez que votre environnement de test génère plus d'informations de débogage que l'environnement de production, vous pouvez définir une variable d'environnement pour configurer

votre environnement de test de manière à utiliser une journalisation plus approfondie ou un suivi plus détaillé.

Par exemple, dans votre environnement de test, vous pouvez définir une variable d'environnement avec la clé `LOG_LEVEL` et une valeur indiquant le niveau de journalisation du débogage ou du suivi. Dans le code de votre fonction Lambda, vous pouvez ensuite utiliser cette variable d'environnement pour définir le niveau de journalisation.

Les exemples de code suivants en Python et Node.js illustrent la façon dont vous pouvez y parvenir. Ces exemples supposent que la valeur de votre variable d'environnement est `DEBUG` en Python, ou `debug` en Node.js.

Python

Example Code Python pour définir le niveau de journalisation

```
import os
import logging

# Initialize the logger
logger = logging.getLogger()

# Get the log level from the environment variable and default to INFO if not set
log_level = os.environ.get('LOG_LEVEL', 'INFO')

# Set the log level
logger.setLevel(log_level)

def lambda_handler(event, context):
    # Produce some example log outputs
    logger.debug('This is a log with detailed debug information - shown only in test
 environment')
    logger.info('This is a log with standard information - shown in production and
 test environments')
```

Node.js (ES module format)

Example Code Node.js pour définir le niveau de journalisation

Cet exemple utilise la bibliothèque de journalisation `winston`. Utilisez npm pour ajouter cette bibliothèque au package de déploiement de votre fonction. Pour de plus amples informations,

veuillez consulter the section called “Création d’un package de déploiement .zip avec dépendances”.

```
import winston from 'winston';

// Initialize the logger using the log level from environment variables, defaulting
 to INFO if not set
const logger = winston.createLogger({
   level: process.env.LOG_LEVEL || 'info',
   format: winston.format.json(),
   transports: [new winston.transports.Console()]
});

export const handler = async (event) => {
   // Produce some example log outputs
   logger.debug('This is a log with detailed debug information - shown only in test
 environment');
   logger.info('This is a log with standard information - shown in production and
 test environment');

};
```

Récupération de variables d’environnement Lambda

Pour récupérer les variables d’environnement dans le code de votre fonction, utilisez la méthode standard pour votre langage de programmation.

Node.js

```
let region = process.env.AWS_REGION
```

Python

```
import os
  region = os.environ['AWS_REGION']
```

> **Note**
>
> Dans certains cas, vous devrez peut-être utiliser le format suivant :

```
region = os.environ.get('AWS_REGION')
```

Ruby

```
region = ENV["AWS_REGION"]
```

Java

```
String region = System.getenv("AWS_REGION");
```

Go

```
var region = os.Getenv("AWS_REGION")
```

C#

```
string region = Environment.GetEnvironmentVariable("AWS_REGION");
```

PowerShell

```
$region = $env:AWS_REGION
```

Lambda stocke les variables d'environnement en toute sécurité en les chiffrant au repos. Vous pouvez configurer Lambda pour utiliser une clé de chiffrement différente, chiffrer les valeurs des variables d'environnement côté client ou définir des variables d'environnement dans un CloudFormation modèle avec. AWS Secrets Manager

Variables d'environnement d'exécution définies

Les runtimes Lambda définissent plusieurs variables d'environnement lors de l'initialisation. La plupart des variables d'environnement fournissent des informations sur la fonction ou l'exécution. Les clés de ces variables d'environnement sont réservées et ne peuvent pas être définies dans la configuration de la fonction.

Variables d'environnement réservées

- `_HANDLER` – Emplacement de gestionnaire configuré sur la fonction.

- `_X_AMZN_TRACE_ID` – En-tête de suivi X-Ray. Cette variable d'environnement change à chaque invocation.
  - Cette variable d'environnement n'est pas définie pour les environnements d'exécution uniquement basés sur le système d'exploitation (famille des environnements d'exécution `provided`). Vous pouvez définir `_X_AMZN_TRACE_ID` pour les environnements d'exécution personnalisés à l'aide de l'en-tête de réponse `Lambda-Runtime-Trace-Id` de l'Invocation suivante.
  - Pour les versions 17 et ultérieures du runtime Java, cette variable d'environnement n'est pas utilisée. Lambda stocke plutôt les informations de suivi dans la propriété du système `com.amazonaws.xray.traceHeader`.
- `AWS_DEFAULT_REGION`— L' Région AWS endroit par défaut où la fonction Lambda est exécutée.
- `AWS_REGION`— L' Région AWS endroit où la fonction Lambda est exécutée. Si elle est définie, cette valeur remplace `AWS_DEFAULT_REGION`.
  - Pour plus d'informations sur l'utilisation des variables d' Région AWS environnement avec AWS SDKs, consultez AWS la section Région dans le guide de référence des outils AWS SDKs et.
- `AWS_EXECUTION_ENV` – Identifiant d'exécution, doté du préfixe `AWS_Lambda_` (par exemple, `AWS_Lambda_java8`). Cette variable d'environnement n'est pas définie pour les environnements d'exécution uniquement basés sur le système d'exploitation (famille des environnements d'exécution `provided`).
- `AWS_LAMBDA_FUNCTION_NAME` – Nom de la fonction.
- `AWS_LAMBDA_FUNCTION_MEMORY_SIZE` – Quantité de mémoire disponible pour la fonction en Mo.
- `AWS_LAMBDA_FUNCTION_VERSION` – Version de la fonction en cours d'exécution.
- `AWS_LAMBDA_INITIALIZATION_TYPE`— Le type d'initialisation de la fonction, qui est `on-demand` `provisioned-concurrency`ou `snap-start`, ou`lambda-managed-instances`. Pour plus d'informations, consultez Configuration de la simultanéité provisionnéeAméliorer les performances de démarrage avec Lambda SnapStart, ou. Instances gérées Lambda
- `AWS_LAMBDA_LOG_GROUP_NAME`, `AWS_LAMBDA_LOG_STREAM_NAME` — Le nom du groupe Amazon CloudWatch Logs et du flux associés à la fonction. Les variables `AWS_LAMBDA_LOG_GROUP_NAME` d'`AWS_LAMBDA_LOG_STREAM_NAME`environnement et ne sont pas disponibles dans les fonctions Lambda SnapStart .
- `AWS_ACCESS_KEY`, `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, `AWS_SESSION_TOKEN` – Clés d'accès obtenues du rôle d'exécution de la fonction.
- `AWS_LAMBDA_RUNTIME_API` – (exécution personnalisée) Hôte et port de l'API d'exécution..

- `LAMBDA_TASK_ROOT` – Chemin d'accès du code de la fonction Lambda.
- `LAMBDA_RUNTIME_DIR` – Chemin d'accès des bibliothèques de l'environnement d'exécution.
- `AWS_LAMBDA_MAX_CONCURRENCY`— (Instances gérées par Lambda uniquement) Le nombre maximum d'appels simultanés que Lambda enverra à un environnement d'exécution.

Les variables d'environnement supplémentaires suivantes ne sont pas réservées et peuvent être étendues dans la configuration de la fonction.

Variables d'environnement non réservées

- `LANG` – Paramètres régionaux de l'environnement d'exécution (`en_US.UTF-8`).
- `PATH` – Chemin d'exécution (`/usr/local/bin:/usr/bin/:/bin:/opt/bin`).
- `LD_LIBRARY_PATH` – Chemin de la bibliothèque système (`/var/lang/lib:/lib64:/usr/lib64:$LAMBDA_RUNTIME_DIR:$LAMBDA_RUNTIME_DIR/lib:$LAMBDA_TASK_ROOT:$LAMBDA_TASK_ROOT/lib:/opt/lib`).
- `NODE_PATH` – (Node.js) Chemin d'accès de la bibliothèque Node.js (`/opt/nodejs/node12/node_modules/:/opt/nodejs/node_modules:$LAMBDA_RUNTIME_DIR/node_modules`).
- `NODE_OPTIONS` – (Node.js) Pour les environnements d'exécution Node.js, vous pouvez utiliser `NODE_OPTIONS` pour réactiver les fonctionnalités expérimentales que Lambda désactive par défaut.
- `PYTHONPATH` : (Python) le chemin de la bibliothèque Python (`$LAMBDA_RUNTIME_DIR`).
- `GEM_PATH` – (Ruby) Chemin d'accès de la bibliothèque Ruby (`$LAMBDA_TASK_ROOT/vendor/bundle/ruby/3.3.0:/opt/ruby/gems/3.3.0`).
- `AWS_XRAY_CONTEXT_MISSING` – Pour le suivi X-Ray, Lambda définit cette valeur sur `LOG_ERROR` pour éviter de générer des erreurs d'environnement d'exécution à partir du kit SDK X-Ray.
- `AWS_XRAY_DAEMON_ADDRESS` – Pour le suivi X-Ray, adresse IP et le port du démon X-Ray.
- `AWS_LAMBDA_DOTNET_PREJIT` : (.NET) définissez cette variable pour activer ou désactiver les optimisations d'environnement d'exécution spécifiques à .NET. Les valeurs incluent `always`, `never` et `provisioned-concurrency`. Pour de plus amples informations, veuillez consulter Configuration de la simultanéité provisionnée pour une fonction.
- `TZ` – Fuseau horaire de l'environnement (`:UTC`). L'environnement d'exécution utilise NTP pour synchroniser l'horloge système.

Les valeurs d'exemple affichées reflètent les derniers moteurs d'exécution. La présence de variables spécifiques ou leurs valeurs peuvent varier dans les anciens environnements d'exécution.

Sécurisation de variables d'environnement Lambda

Pour sécuriser vos variables d'environnement, vous pouvez utiliser le chiffrement côté serveur pour protéger vos données au repos, et le chiffrement côté client pour protéger vos données en transit.

> **Note**
>
> Pour renforcer la sécurité de la base de données, nous vous recommandons d'utiliser à la AWS Secrets Manager place des variables d'environnement pour stocker les informations d'identification de la base de données. Pour de plus amples informations, veuillez consulter Utiliser des secrets Secrets Manager dans des fonctions Lambda.

Sécurité au repos

Lambda fournit toujours le chiffrement côté serveur au repos grâce à une AWS KMS key. Par défaut, Lambda utilise une Clé gérée par AWS. Si ce comportement par défaut convient à votre flux de travail, vous n'avez pas besoin de configurer quoi que ce soit d'autre. Lambda les crée Clé gérée par AWS dans votre compte et gère les autorisations pour vous. AWS l'utilisation de cette clé ne vous est pas facturée.

Si vous préférez, vous pouvez plutôt fournir une clé gérée par le AWS KMS client. Vous pouvez procéder ainsi pour contrôler la rotation de la clé KMS ou pour répondre aux exigences de votre organisation en matière de gestion des clés KMS. Si vous utilisez une clé gérée par le client, seuls les utilisateurs de votre compte ayant accès à la clé KMS peuvent afficher ou gérer les variables d'environnement sur la fonction.

Les clés gérées par le client entraînent des AWS KMS frais standard. Pour en savoir plus, consultez Pricing AWS Key Management Service (Tarification).

Sécurité pendant le transit

Pour plus de sécurité, vous pouvez activer les assistants pour le chiffrement en transit, ce qui garantit que vos variables d'environnement sont chiffrées côté client pour une meilleure protection pendant le transit.

Pour configurer le chiffrement de vos variables d'environnement

1. Utilisez le AWS Key Management Service (AWS KMS) pour créer des clés gérées par le client que Lambda utilisera pour le chiffrement côté serveur et côté client. Pour en savoir plus, consultez Création des clés dans le AWS Key Management Service Guide du développeur .
2. À l'aide de la console Lambda, accédez à la page Modifier les variables d'environnement.
   a. Ouvrez la page Functions (Fonctions) de la console Lambda.
   b. Choisissez une fonction.
   c. Choisissez Configuration, puis Variables d'environnement dans la barre de navigation de gauche.
   d. Dans la section Variables d'environnement), choisissez Modifier.
   e. Développez Configuration du chiffrement.
3. (Facultatif) Activez les assistants de chiffrement de la console afin qu'ils utilisent le chiffrement côté client pour protéger vos données en transit.
   a. Sous Encryption in transit (Chiffrement pendant le transit), choisissez Enable helpers for encryption in transit (Activer les assistants pour le chiffrement pendant le transit).
   b. Pour chaque variable d'environnement pour laquelle vous souhaitez activer les assistants de chiffrement de la console, choisissez Encrypt (Chiffrer) en regard de la variable d'environnement.
   c. Sous AWS KMS key Pour chiffrer en cours de transfert, choisissez une clé gérée par le client que vous avez créée au début de cette procédure.
   d. Choisissez Politique de rôle d'exécution et copiez la politique. Cette politique octroie au rôle d'exécution de votre fonction l'autorisation de déchiffrer les variables d'environnement.

      Enregistrez cette politique, car vous l'utiliserez à la dernière étape de cette procédure.
   e. Ajoutez le code à votre fonction qui déchiffre les variables d'environnement . Pour obtenir un exemple, choisissez Déchiffrer l'extrait de secrets.
4. (Facultatif) Spécifiez votre clé gérée par le client pour le chiffrement au repos.
   a. Choisissez Utiliser une clé principale client.
   b. Choisissez une clé gérée par le client que vous avez créée au début de cette procédure.
5. Choisissez Enregistrer.
6. Configurez des autorisations.

Si vous utilisez une clé gérée par le client avec chiffrement côté serveur, octroyez des autorisations à tous les utilisateurs ou rôles que vous voulez pouvoir afficher ou pour gérer des variables d'environnement sur la fonction. Pour de plus amples informations, veuillez consulter Gestion des autorisations sur votre clé de chiffrement KMS côté serveur.

Si vous activez le chiffrement côté client pour la sécurité pendant le transit, votre fonction doit disposer d'une autorisation pour appeler l'opération d'API `kms:Decrypt`. Ajoutez la politique que vous avez précédemment enregistrée dans cette procédure au rôle d'exécution de la fonction.

Gestion des autorisations sur votre clé de chiffrement KMS côté serveur

Aucune AWS KMS autorisation n'est requise pour que votre utilisateur ou le rôle d'exécution de la fonction utilise la clé de chiffrement par défaut. Pour utiliser une clé gérée par le client, vous devez disposer des autorisations appropriées pour utiliser la clé. Lambda utilise vos autorisations pour créer un octroi sur la clé. Cela permet à Lambda de l'utiliser pour le chiffrement.

- `kms:ListAliases` – Pour afficher les clés dans la console Lambda.
- `kms:CreateGrant`, `kms:Encrypt` – Pour configurer une clé gérée par le client sur une fonction.
- `kms:Decrypt` – Pour afficher et gérer des variables d'environnement chiffrées à l'aide d'une clé gérée par le client.

Vous pouvez obtenir ces autorisations à partir de votre politique d'autorisation basée sur les ressources Compte AWS ou de celle d'une clé. `ListAliasesest` fourni par les politiques gérées pour Lambda. Les stratégies de clé accordent les autorisations restantes aux utilisateurs du groupe Key users (Utilisateurs de clés).

Les utilisateurs sans autorisation `Decrypt` peuvent gérer les fonctions, mais ils ne peuvent pas afficher les variables d'environnement ni les gérer dans la console Lambda. Pour empêcher un utilisateur d'afficher des variables d'environnement, ajoutez une instruction aux autorisations de cet utilisateur afin de refuser l'accès à la clé par défaut, à une clé gérée par le client ou à toutes les clés.

Example Politique IAM – Refuser l'accès en fonction de l'ARN de la clé

JSON

```
{
```

```
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Deny",
            "Action": [
                "kms:Decrypt"
            ],
            "Resource": "arn:aws:kms:us-east-2:111122223333:key/3be10e2d-
xmpl-4be4-bc9d-0405a71945cc"
        }
    ]
}
```

Pour en savoir plus sur la gestion des autorisations de clés, consultez Stratégies de clé dans AWS KMS que vous trouverez dans le Guide du développeur AWS Key Management Service .

Octroi aux fonctions Lambda d’un accès aux ressources d’un Amazon VPC

Avec Amazon Virtual Private Cloud (Amazon VPC), vous pouvez créer des réseaux privés Compte AWS pour héberger des ressources telles que des instances Amazon Elastic Compute Cloud (Amazon EC2), des instances Amazon Relational Database Service (Amazon RDS) et des instances Amazon. ElastiCache Vous pouvez donner à votre fonction Lambda l’accès aux ressources hébergées dans un Amazon VPC en attachant votre fonction au VPC via les sous-réseaux privés qui contiennent les ressources. Suivez les instructions des sections suivantes pour associer une fonction Lambda à un Amazon VPC à l'aide de la console Lambda, du () ou. AWS Command Line Interface AWS CLI AWS SAM

> **Note**
>
> Chaque fonction Lambda s’exécute au sein d’un VPC détenu et géré par le service Lambda. Ils VPCs sont gérés automatiquement par Lambda et ne sont pas visibles pour les clients. La configuration de votre fonction pour accéder à d'autres AWS ressources d'un Amazon VPC n'a aucun effet sur le VPC géré par Lambda dans lequel votre fonction exécute votre fonction.

Sections

- Autorisations IAM requises
- Associer des fonctions Lambda à un Amazon VPC dans votre Compte AWS
- Accès à Internet en cas d’attachement à un VPC
- IPv6 soutien
- Bonnes pratiques d'utilisation de Lambda avec Amazon VPCs
- Comprendre les interfaces réseau élastiques Hyperplane () ENIs
- Utilisation des clés de condition IAM pour les paramètres du VPC
- Didacticiels de VPC

Autorisations IAM requises

Pour associer une fonction Lambda à un Amazon VPC dans votre VPC, Compte AWS Lambda a besoin d'autorisations pour créer et gérer les interfaces réseau qu'il utilise pour permettre à votre fonction d'accéder aux ressources du VPC.

Les interfaces réseau créées par Lambda sont appelées Hyperplane Elastic Network Interfaces ou Hyperplane. ENIs Pour en savoir plus sur ces interfaces réseau, consultez the section called “Comprendre les interfaces réseau élastiques Hyperplane () ENIs”.

Vous pouvez accorder à votre fonction les autorisations dont elle a besoin en associant la politique AWS gérée AWSLambdaVPCAccessExecutionRoleau rôle d'exécution de votre fonction. Lorsque vous créez une fonction dans la console Lambda et que vous l’associez à un VPC, Lambda ajoute automatiquement cette politique d’autorisations pour vous.

Si vous préférez créer votre propre politique d’autorisations IAM, veillez à inclure toutes les autorisations suivantes et à les autoriser sur toutes les ressources (`"Resource": "*"`) :

- EC2 : CreateNetworkInterface
- EC2 : DescribeNetworkInterfaces
- EC2 : DescribeSubnets
- EC2 : DeleteNetworkInterface
- EC2 : AssignPrivateIpAddresses
- EC2 : UnassignPrivateIpAddresses

Notez que le rôle de votre fonction n’a besoin de ces autorisations que pour créer les interfaces réseau, et non pour invoquer votre fonction. Votre fonction peut toujours être invoquée avec succès lorsqu’elle est attachée à un VPC Amazon, même si vous supprimez ces autorisations du rôle d’exécution de votre fonction.

Pour associer votre fonction à un VPC, Lambda doit également vérifier les ressources réseau à l’aide de votre rôle d’utilisateur IAM. Veillez à ce que votre rôle d’utilisateur dispose des autorisations IAM suivantes :

- EC2 : DescribeSecurityGroups
- EC2 : DescribeSubnets
- EC2 : DescribeVpcs
- EC2 : GetSecurityGroupsForVpc

> **Note**
>
> Les EC2 autorisations Amazon que vous accordez au rôle d'exécution de votre fonction sont utilisées par le service Lambda pour associer votre fonction à un VPC. Cependant, ces autorisations sont également accordées implicitement au code de votre fonction. Cela signifie que votre code de fonction est capable d'effectuer ces appels d' EC2 API Amazon. Pour obtenir des conseils sur le respect des pratiques exemplaires en matière de sécurité, consultez the section called “Bonnes pratiques de sécurité”.

Associer des fonctions Lambda à un Amazon VPC dans votre Compte AWS

Associez votre fonction à un Amazon VPC dans votre ordinateur à l'aide Compte AWS de la console Lambda, du ou. AWS CLI AWS SAM Si vous utilisez le AWS CLI ou AWS SAM, ou si vous attachez une fonction existante à un VPC à l'aide de la console Lambda, assurez-vous que le rôle d'exécution de votre fonction dispose des autorisations nécessaires répertoriées dans la section précédente.

Les fonctions Lambda ne peuvent pas se connecter directement à un VPC avec la location d’instance dédiée. Pour vous connecter à des ressources dans un VPC dédié, associez-les à un deuxième VPC avec une location par défaut.

Lambda console

Pour attacher une fonction lors de sa création à un Amazon VPC

1. Ouvrez la page Fonctions de la console Lambda et choisissez Créer une fonction.
2. Sous Informations de base, dans Nom de fonction, entrez un nom pour votre fonction.
3. Configurez les paramètres VPC pour la fonction en procédant comme suit :
   a. Développez Advanced settings (Paramètres avancés).
   b. Sélectionnez Activer le VPC, puis choisissez le VPC auquel vous souhaitez associer la fonction.
   c. (Facultatif) Pour autoriser le IPv6 trafic sortant, sélectionnez Autoriser le IPv6 trafic pour les sous-réseaux à double pile.
   d. Choisissez les sous-réseaux et les groupes de sécurité pour lesquels créer l’interface réseau. Si vous avez sélectionné Autoriser IPv6 le trafic pour les sous-réseaux à double pile, tous les sous-réseaux sélectionnés doivent comporter un bloc IPv4 CIDR et un bloc CIDR. IPv6

> **Note**
>
> Pour accéder aux ressources privées, connectez votre fonction à des sous-réseaux privés. Si votre fonction a besoin d'un accès Internet, utilisez the section called "Accès Internet pour les fonctions VPC". La connexion d'une fonction à un sous-réseau public ne lui donne pas accès à Internet ou à une adresse IP publique.

4. Sélectionnez Create function (Créer une fonction).

Pour associer une fonction existante à un Amazon VPC

1. Ouvrez la page Fonctions de la console Lambda et choisissez votre fonction.
2. Choisissez l'onglet Configuration, puis VPC.
3. Choisissez Modifier.
4. Sous VPC, sélectionnez l'Amazon VPC auquel vous souhaitez associer votre fonction.
5. (Facultatif) Pour autoriser le IPv6 trafic sortant, sélectionnez Autoriser le IPv6 trafic pour les sous-réseaux à double pile.
6. Choisissez les sous-réseaux et les groupes de sécurité pour lesquels créer l'interface réseau. Si vous avez sélectionné Autoriser IPv6 le trafic pour les sous-réseaux à double pile, tous les sous-réseaux sélectionnés doivent comporter un bloc IPv4 CIDR et un bloc CIDR. IPv6

   > **Note**
   >
   > Pour accéder aux ressources privées, connectez votre fonction à des sous-réseaux privés. Si votre fonction a besoin d'un accès Internet, utilisez the section called "Accès Internet pour les fonctions VPC". La connexion d'une fonction à un sous-réseau public ne lui donne pas accès à Internet ou à une adresse IP publique.

7. Choisissez Enregistrer.

AWS CLI

Pour attacher une fonction lors de sa création à un Amazon VPC

- Pour créer une fonction Lambda et l'attacher à un VPC, exécutez la commande `create-function` de la CLI suivante.

```
aws lambda create-function --function-name my-function \
--runtime nodejs24.x --handler index.js --zip-file fileb://function.zip \
--role arn:aws:iam::123456789012:role/lambda-role \
--vpc-config
 Ipv6AllowedForDualStack=true,SubnetIds=subnet-071f712345678e7c8,subnet-07fd123456788a03
```

Spécifiez vos propres sous-réseaux et groupes de sécurité et configurez `Ipv6AllowedForDualStack` sur `true` ou `false` en fonction de votre cas d'utilisation.

Pour associer une fonction existante à un Amazon VPC

- Pour attacher une fonction existante à un VPC, exécutez la commande `update-function-configuration` de la CLI suivante.

```
aws lambda update-function-configuration --function-name my-function \
--vpc-config Ipv6AllowedForDualStack=true,
 SubnetIds=subnet-071f712345678e7c8,subnet-07fd123456788a036,SecurityGroupIds=sg-0859123
```

Pour dissocier votre fonction d'un VPC

- Pour dissocier votre fonction d'un VPC, exécutez la commande `update-function-configuration` de la CLI suivante avec une liste vide de sous-réseaux et de groupes de sécurité VPC.

```
aws lambda update-function-configuration --function-name my-function \
--vpc-config SubnetIds=[],SecurityGroupIds=[]
```

AWS SAM

Pour attacher votre fonction à un VPC

- Pour associer une fonction Lambda à un Amazon VPC, ajoutez la propriété `VpcConfig` à votre définition de fonction, comme illustré par l'exemple de modèle suivant. Pour plus d'informations sur cette propriété, voir AWS: :Lambda : :Function VpcConfig dans le guide de CloudFormation l'utilisateur (la AWS SAM `VpcConfig` propriété est transmise directement à la `VpcConfig` propriété d'une CloudFormation `AWS::Lambda::Function` ressource).

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31

Resources:
  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./lambda_function/
      Handler: lambda_function.handler
      Runtime: python3.12
      VpcConfig:
        SecurityGroupIds:
          - !Ref MySecurityGroup
        SubnetIds:
          - !Ref MySubnet1
          - !Ref MySubnet2
      Policies:
        - AWSLambdaVPCAccessExecutionRole

  MySecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: Security group for Lambda function
      VpcId: !Ref MyVPC

  MySubnet1:
    Type: AWS::EC2::Subnet
    Properties:
      VpcId: !Ref MyVPC
      CidrBlock: 10.0.1.0/24

  MySubnet2:
    Type: AWS::EC2::Subnet
```

```
    Properties:
      VpcId: !Ref MyVPC
      CidrBlock: 10.0.2.0/24

  MyVPC:
    Type: AWS::EC2::VPC
    Properties:
      CidrBlock: 10.0.0.0/16
```

Pour plus d'informations sur la configuration de votre VPC dans AWS SAM, consultez AWS: : EC2 : :VPC dans le guide de l'utilisateur.CloudFormation

Accès à Internet en cas d’attachement à un VPC

Par défaut, les fonctions Lambda ont accès à l’Internet public. Lorsque vous attachez votre fonction à un VPC, son accès est limité aux ressources disponibles au sein de ce VPC. Pour accorder à votre fonction l’accès à Internet, vous devez également configurer le VPC pour qu’il dispose d’un accès à Internet. Pour en savoir plus, veuillez consulter la section the section called “Accès Internet pour les fonctions VPC”.

IPv6 soutien

Votre fonction peut se connecter à des ressources dans des sous-réseaux VPC à double pile via. IPv6 Cette option est désactivée par défaut. Pour autoriser le IPv6 trafic sortant, utilisez la console ou l'`--vpc-config Ipv6AllowedForDualStack=true`option associée à la fonction ou à la commande de création. update-function-configuration

> **Note**
>
> Pour autoriser le IPv6 trafic sortant dans un VPC, tous les sous-réseaux connectés à la fonction doivent être des sous-réseaux à double pile. Lambda ne prend pas en charge les IPv6 connexions IPv6 sortantes pour les sous-réseaux uniquement d'un VPC ni les connexions IPv6 sortantes pour les fonctions qui ne sont pas connectées à un VPC.

Vous pouvez mettre à jour votre code de fonction pour vous connecter explicitement aux ressources du sous-réseau. IPv6 L'exemple Python suivant ouvre un socket et se connecte à un IPv6 serveur.

Example — Se connecter au IPv6 serveur

```
def connect_to_server(event, context):
    server_address = event['host']
    server_port = event['port']
    message = event['message']
    run_connect_to_server(server_address, server_port, message)

def run_connect_to_server(server_address, server_port, message):
    sock = socket.socket(socket.AF_INET6, socket.SOCK_STREAM, 0)
    try:
        # Send data
        sock.connect((server_address, int(server_port), 0, 0))
        sock.sendall(message.encode())
        BUFF_SIZE = 4096
        data = b''
        while True:
            segment = sock.recv(BUFF_SIZE)
            data += segment
            # Either 0 or end of data
            if len(segment) < BUFF_SIZE:
                break
        return data
    finally:
        sock.close()
```

Bonnes pratiques d'utilisation de Lambda avec Amazon VPCs

Pour vous assurer que votre configuration VPC Lambda est conforme aux pratiques exemplaires en la matière, suivez les conseils des sections suivantes.

Bonnes pratiques de sécurité

Pour associer votre fonction Lambda à un VPC, vous devez attribuer un certain nombre d'autorisations Amazon au rôle d'exécution de votre fonction. EC2 Ces autorisations sont nécessaires pour créer les interfaces réseau que votre fonction utilise pour accéder aux ressources du VPC. Cependant, ces autorisations sont également octroyées implicitement au code de votre fonction. Cela signifie que votre code de fonction est autorisé à effectuer ces appels d' EC2 API Amazon.

Pour respecter le principe de l’accès au moindre privilège, ajoutez une politique de rejet au rôle d’exécution de votre fonction, comme dans l’exemple suivant. Cette politique empêche votre code de

fonction d'appeler Amazon EC2 APIs, tout en permettant au service Lambda de gérer les ressources VPC en votre nom. La politique utilise la clé de condition `lambda:SourceFunctionArn`, qui s'applique uniquement aux appels d'API effectués par le code de votre fonction lors de l'exécution. Pour de plus amples informations, veuillez consulter the section called "ARN de la fonction source".

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                 "ec2:CreateNetworkInterface",
                 "ec2:DeleteNetworkInterface",
                 "ec2:DescribeNetworkInterfaces",
                 "ec2:DescribeSubnets",
                 "ec2:DetachNetworkInterface",
                 "ec2:AssignPrivateIpAddresses",
                 "ec2:UnassignPrivateIpAddresses"
            ],
            "Resource": [ "*" ],
            "Condition": {
                "ArnEquals": {
                    "lambda:SourceFunctionArn": [
                        "arn:aws:lambda:us-
west-2:123456789012:function:my_function"
                    ]
                }
            }
        }
    ]
}
```

AWS fournit des groupes de sécurité et des listes de contrôle d'accès réseau (ACLs) pour renforcer la sécurité de votre VPC. Les groupes de sécurité contrôlent le trafic entrant et sortant pour vos ressources, et le réseau ACLs contrôlent le trafic entrant et sortant pour vos sous-réseaux. Les groupes de sécurité offrent un contrôle d'accès suffisant pour la plupart des sous-réseaux. Vous pouvez utiliser le réseau ACLs si vous souhaitez ajouter une couche de sécurité supplémentaire à votre VPC. Pour obtenir des directives générales sur les meilleures pratiques de sécurité lors de

l'utilisation d'Amazon VPCs, consultez les meilleures pratiques de sécurité pour votre VPC dans le guide de l'utilisateur d'Amazon Virtual Private Cloud.

Bonnes pratiques en matière de performances

Lorsque vous attachez votre fonction à un VPC, Lambda vérifie s'il existe une ressource réseau disponible (ENI Hyperplane) à laquelle elle peut se connecter. Les hyperplans ENIs sont associés à une combinaison particulière de groupes de sécurité et de sous-réseaux VPC. Si vous avez déjà attaché une fonction à un VPC, le fait de spécifier les mêmes sous-réseaux et groupes de sécurité lorsque vous attachez une autre fonction signifie que Lambda peut partager les ressources du réseau et éviter d'avoir à créer une ENI Hyperplane. Pour plus d'informations sur Hyperplane ENIs et son cycle de vie, consultezthe section called "Comprendre les interfaces réseau élastiques Hyperplane () ENIs".

Comprendre les interfaces réseau élastiques Hyperplane () ENIs

Une ENI Hyperplane est une ressource gérée qui agit comme une interface réseau entre votre fonction Lambda et les ressources auxquelles vous souhaitez que votre fonction se connecte. Le service Lambda les crée et les gère ENIs automatiquement lorsque vous attachez votre fonction à un VPC.

Les hyperplans ne ENIs sont pas directement visibles pour vous et vous n'avez pas besoin de les configurer ou de les gérer. Cependant, être au fait de leur fonctionnement peut vous aider à comprendre le comportement de votre fonction lorsque vous l'attachez à un VPC.

La première fois que vous attachez une fonction à un VPC à l'aide d'une combinaison particulière de sous-réseau et de groupe de sécurité, Lambda crée une ENI Hyperplane. D'autres fonctions de votre compte qui utilisent la même combinaison de sous-réseau et de groupe de sécurité peuvent également utiliser cette ENI. Dans la mesure du possible, Lambda réutilise les ressources existantes ENIs pour optimiser l'utilisation des ressources et minimiser la création de nouvelles ressources. ENIs Cependant, chaque ENI Hyperplane prend en charge jusqu'à 65 000 connexions/ports. Si le nombre de connexions dépasse cette limite, Lambda adapte le nombre de connexions ENIs automatiquement en fonction du trafic réseau et des exigences de simultanéité.

Pour les nouvelles fonctions, pendant que Lambda crée une ENI Hyperplane, votre fonction reste à l'état En attente et vous ne pouvez pas l'invoquer. Votre fonction passe à l'état Actif uniquement lorsque l'ENI Hyperplane est prête, ce qui peut prendre plusieurs minutes. Pour les fonctions existantes, aucune opération supplémentaire ciblant la fonction ne peut être effectuée, comme la

création de versions ou la mise à jour du code de la fonction, mais vous pouvez continuer à invoquer les versions antérieures de la fonction.

Dans le cadre de la gestion du cycle de vie d'ENI, Lambda peut supprimer et recréer pour équilibrer la charge du trafic réseau ENIs ou ENIs pour résoudre les problèmes détectés lors des bilans de santé d'ENI. De plus, si une fonction Lambda reste inactive pendant 14 jours, Lambda récupère tout hyperplan inutilisé ENIs et définit l'état de la fonction sur. `Inactive` La prochaine tentative d'invocation échouera et la fonction entrera à nouveau dans l'état En attente jusqu'à ce que Lambda termine la création ou l'allocation d'une ENI Hyperplane. Nous recommandons que votre design ne repose pas sur la persistance de ENIs.

Lorsque vous mettez à jour une fonction pour supprimer sa configuration VPC, Lambda a besoin de 20 minutes au maximum pour supprimer l'ENI Hyperplane attaché. Lambda ne supprime l'ENI que si aucune autre fonction (ou version de fonction publiée) n'utilise cette ENI Hyperplane.

Lambda s'appuie sur les permissions du rôle d'exécution de la fonction pour supprimer l'ENI Hyperplane. Si vous supprimez le rôle d'exécution avant que Lambda ne supprime l'ENI Hyperplane, Lambda ne pourra pas la supprimer. Vous pouvez effectuer la suppression manuellement.

Utilisation des clés de condition IAM pour les paramètres du VPC

Vous pouvez utiliser des clés de condition spécifiques de Lambda pour les paramètres du VPC afin de fournir des contrôles d'autorisation supplémentaires pour vos fonctions Lambda. Par exemple, vous pouvez exiger que toutes les fonctions de votre organisation soient connectées à un VPC. Vous pouvez également spécifier les sous-réseaux et les groupes de sécurité que les utilisateurs de la fonction peuvent et ne peuvent pas utiliser.

Lambda prend également en charge les clés de condition suivantes dans les stratégies IAM :

- lambda : VpcIds — Autoriser ou refuser un ou plusieurs VPCs.
- lambda : SubnetIds — Autoriser ou refuser un ou plusieurs sous-réseaux.
- lambda : SecurityGroupIds — Autoriser ou refuser un ou plusieurs groupes de sécurité.

L'API Lambda fonctionne CreateFunctionet UpdateFunctionConfigurationprend en charge ces clés de condition. Pour de plus amples informations sur l'utilisation de clés de condition dans des stratégies IAM, consultez Éléments de politique JSON IAM : Condition dans le Guide de l'utilisateur IAM.

> **Tip**
>
> Si votre fonction inclut déjà une configuration VPC à partir d'une demande d'API précédente, vous pouvez envoyer une demande `UpdateFunctionConfiguration` sans la configuration du VPC.

Exemple de stratégies avec des clés de condition pour les paramètres du VPC

Les exemples suivants montrent comment utiliser les clés de condition pour les paramètres du VPC. Après avoir créé une instruction de politique avec les restrictions souhaitées, ajoutez l'instruction de politique pour l'utilisateur ou le rôle cible.

Assurez-vous que les utilisateurs déploient uniquement les fonctions connectées au VPC

Pour vous assurer que tous les utilisateurs déploient uniquement des fonctions connectées au VPC, vous pouvez refuser les opérations de création et de mise à jour de fonctions qui n'incluent pas d'ID de VPC valide.

Notez que l'ID de VPC n'est pas un paramètre d'entrée pour la demande `CreateFunction` ou `UpdateFunctionConfiguration`. Lambda récupère la valeur de l'ID de VPC en fonction des paramètres du sous-réseau et du groupe de sécurité.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "EnforceVPCFunction",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "Null": {
           "lambda:VpcIds": "true"
        }
      }
```

```
    }
  ]
}
```

Refuser aux utilisateurs l'accès à VPCs des sous-réseaux ou à des groupes de sécurité spécifiques

Pour refuser aux utilisateurs l'accès à une condition spécifique VPCs, utilisez cette option `StringEquals` pour vérifier la valeur de la `lambda:VpcIds` condition. L'exemple suivant refuse aux utilisateurs l'accès à `vpc-1` et `vpc-2`.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "EnforceOutOfVPC",
            "Action": [
                "lambda:CreateFunction",
                "lambda:UpdateFunctionConfiguration"
            ],
            "Effect": "Deny",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "lambda:VpcIds": [
                        "vpc-1",
                        "vpc-2"
                    ]
                }
            }
        }
    ]
}
```

Pour refuser aux utilisateurs l'accès à des sous-réseaux spécifiques, utilisez `StringEquals` pour vérifier la valeur de la condition `lambda:SubnetIds`. L'exemple suivant refuse aux utilisateurs l'accès à `subnet-1` et `subnet-2`.

```
{
      "Sid": "EnforceOutOfSubnet",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "ForAnyValue:StringEquals": {
            "lambda:SubnetIds": ["subnet-1", "subnet-2"]
        }
      }
    }
```

Pour refuser aux utilisateurs l'accès à des groupes de sécurité spécifiques, utilisez `StringEquals` pour vérifier la valeur de la condition `lambda:SecurityGroupIds`. L'exemple suivant refuse aux utilisateurs l'accès à `sg-1` et `sg-2`.

```
{
      "Sid": "EnforceOutOfSecurityGroups",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "ForAnyValue:StringEquals": {
            "lambda:SecurityGroupIds": ["sg-1", "sg-2"]
        }
      }
    }
  ]
}
```

Autoriser les utilisateurs à créer et à mettre à jour des fonctions avec des paramètres VPC spécifiques

Pour autoriser les utilisateurs à accéder à une condition spécifique VPCs, utilisez cette option `StringEquals` pour vérifier la valeur de la `lambda:VpcIds` condition. L'exemple suivant permet aux utilisateurs d'accéder à `vpc-1` et `vpc-2`.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "EnforceStayInSpecificVpc",
            "Action": [
                "lambda:CreateFunction",
                "lambda:UpdateFunctionConfiguration"
            ],
            "Effect": "Allow",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "lambda:VpcIds": [
                        "vpc-1",
                        "vpc-2"
                    ]
                }
            }
        }
    ]
}
```

Pour permettre aux utilisateurs d'accéder à des sous-réseaux spécifiques, utilisez `StringEquals` pour vérifier la valeur de la condition `lambda:SubnetIds`. L'exemple suivant permet aux utilisateurs d'accéder à `subnet-1` et `subnet-2`.

```
{
      "Sid": "EnforceStayInSpecificSubnets",
      "Action": [
          "lambda:CreateFunction",
```

```
            "lambda:UpdateFunctionConfiguration"
        ],
        "Effect": "Allow",
        "Resource": "*",
        "Condition": {
          "ForAllValues:StringEquals": {
              "lambda:SubnetIds": ["subnet-1", "subnet-2"]
          }
        }
    }
```

Pour permettre aux utilisateurs d'accéder à des groupes de sécurité spécifiques, utilisez `StringEquals` pour vérifier la valeur de la condition `lambda:SecurityGroupIds`. L'exemple suivant permet aux utilisateurs d'accéder à `sg-1` et `sg-2`.

```
{
      "Sid": "EnforceStayInSpecificSecurityGroup",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Allow",
      "Resource": "*",
      "Condition": {
        "ForAllValues:StringEquals": {
            "lambda:SecurityGroupIds": ["sg-1", "sg-2"]
        }
      }
    }
  ]
}
```

Didacticiels de VPC

Dans les didacticiels suivants, vous connectez une fonction Lambda à des ressources dans votre VPC.

- Tutoriel : Utilisation d'une fonction Lambda pour accéder à Amazon RDS dans un Amazon VPC

- Tutoriel : Configuration d'une fonction Lambda pour accéder à Amazon ElastiCache dans un Amazon VPC

Octroi aux fonctions Lambda d’un accès à une ressource dans un Amazon VPC d’un autre compte

Vous pouvez donner à votre AWS Lambda fonction l'accès à une ressource d'un Amazon VPC dans Amazon Virtual Private Cloud géré par un autre compte, sans exposer l'un ou l'autre VPC à Internet. Ce modèle d'accès vous permet de partager des données avec d'autres organisations utilisant AWS. Grâce à ce modèle d'accès, vous pouvez partager des données VPCs avec un niveau de sécurité et de performance supérieur à celui d'Internet. Configurez votre fonction Lambda pour utiliser une connexion d’appairage d’Amazon VPC pour accéder à ces ressources.

> **Warning**
>
> Lorsque vous autorisez l'accès entre les comptes ou VPCs que vous vérifiez que votre plan répond aux exigences de sécurité des organisations respectives qui gèrent ces comptes. Le respect des instructions de ce document aura une incidence sur le niveau de sécurité de vos ressources.

Dans ce didacticiel, vous connectez deux comptes à l'aide d'une connexion d'appairage à l'aide IPv4 de. Vous configurez une fonction Lambda qui n’est pas déjà connectée à un Amazon VPC. Vous configurez la résolution DNS pour connecter votre fonction à des ressources qui ne fournissent pas de données statiques IPs. Pour adapter ces instructions à d’autres scénarios d’appairage, consultez le Guide d’appairage de VPC.

Conditions préalables

Pour permettre à une fonction Lambda d’accéder à une ressource d’un autre compte, vous devez avoir :

- une fonction Lambda, configurée pour s’authentifier auprès de votre ressource puis lire à partir de celle-ci ;
- une ressource d’un autre compte, tel qu’un cluster Amazon RDS, disponible via Amazon VPC ;
- les informations d’identification du compte de votre fonction Lambda et du compte de votre ressource. Si vous n’êtes pas autorisé à utiliser le compte de votre ressource, contactez un utilisateur autorisé pour préparer ce compte ;
- l’autorisation de créer et de mettre à jour un VPC (et de prendre en charge les ressources Amazon VPC) à associer à votre fonction Lambda ;

- l’autorisation de mettre à jour le rôle d’exécution et la configuration VPC de votre fonction Lambda ;
- l’autorisation de créer une connexion d’appairage de VPC dans le compte de votre fonction Lambda ;
- l’autorisation d’accepter une connexion d’appairage de VPC dans le compte de votre ressource ;
- l’autorisation de mettre à jour la configuration du VPC de votre ressource (et de prendre en charge les ressources Amazon VPC) ;
- l’autorisation d’invoquer votre fonction Lambda.

Création d’un Amazon VPC dans le compte de votre fonction

Créez un Amazon VPC, des sous-réseaux, des tables de routage et un groupe de sécurité dans le compte de votre fonction Lambda.

Pour créer un VPC, des sous-réseaux et d'autres ressources VPC à l'aide de la console

1. Ouvrez la console Amazon VPC à l’adresse https://console.aws.amazon.com/vpc/.
2. Sur le tableau de bord, choisissez Créer un VPC.
3. Pour le bloc IPv4 CIDR, fournissez un bloc CIDR privé. Votre bloc d’adresse CIDR ne doit chevaucher aucun bloc utilisé dans le VPC de votre ressource. Ne choisissez pas un bloc que le VPC de vos ressources utilise pour l'attribuer IPs aux ressources ou un bloc déjà défini dans les tables de routage de votre VPC de ressources. Pour plus d’informations sur la définition des blocs d’adresse CIDR appropriés, consultez VPC CIDR blocks.
4. Choisissez Personnaliser AZs.
5. Sélectionnez la même ressource AZs que votre ressource.
6. Pour Nombre de sous-réseaux publics, choisissez 0.
7. Pour VPC endpoints (Points de terminaison d'un VPC), choisissez None (Aucun).
8. Sélectionnez Create VPC (Créer un VPC).

Octroi d’autorisations VPC au rôle d’exécution de votre fonction

AWSLambdaVPCAccessExecutionRoleAttachez-vous au rôle d'exécution de votre fonction pour lui permettre de se connecter à VPCs.

Pour octroyer des autorisations VPC au rôle d'exécution de votre fonction

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom de votre fonction .
3. Choisissez Configuration.
4. Choisissez Autorisations.
5. Sous Nom du rôle, choisissez le rôle d'exécution.
6. Dans la section Politiques d'autorisations, choisissez Ajouter des autorisations.
7. Dans la liste déroulante, choisissez Attacher des politiques.
8. Dans la zone de recherche, saisissez `AWSLambdaVPCAccessExecutionRole`.
9. À gauche du nom de politique, cochez la case.
10. Choisissez Ajouter des autorisations.

Pour attacher votre fonction à votre Amazon VPC

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom de votre fonction .
3. Choisissez l'onglet Configuration, puis VPC.
4. Choisissez Modifier.
5. Sous VPC, sélectionnez votre VPC.
6. Sous Sous-réseaux, choisissez vos sous-réseaux.
7. Sous Groupe de sécurité, sélectionnez le groupe de sécurité par défaut de votre VPC.
8. Choisissez Enregistrer.

Création d'une requête de connexion d'appairage de VPC

Créez une requête de connexion d'appairage de VPC entre le VPC de votre fonction (le VPC demandeur) et le VPC de votre ressource (le VPC accepteur).

Pour demander une connexion d'appairage de VPC depuis le VPC de votre fonction

1. Ouvrez la https://console.aws.amazon.com/vpc/.
2. Dans le volet de navigation, choisissez Peering connections (Connexions d'appairage).
3. Choisissez Create peering connection (Créer une connexion d'appairage).

4. Pour ID VPC (demandeur), sélectionnez le VPC de votre fonction.
5. Pour ID de compte, saisissez l'ID du compte de votre ressource.
6. Pour VPC ID (Accepter), saisissez le VPC de votre ressource.

Préparation du compte de votre ressource

Pour créer votre connexion d'appairage et préparer le VPC de votre ressource à utiliser la connexion, connectez-vous au compte de votre ressource avec un rôle doté des autorisations répertoriées dans les conditions requises. Les étapes de connexion peuvent être différentes en fonction de la manière dont le compte est sécurisé. Pour plus d'informations sur la façon de se connecter à un AWS compte, consultez le Guide de l'utilisateur de AWS connexion. Au sein du compte de votre ressource, effectuez les procédures suivantes.

Pour accepter une requête de connexion d'appairage de VPC

1. Ouvrez la https://console.aws.amazon.com/vpc/.
2. Dans le volet de navigation, choisissez Peering connections (Connexions d'appairage).
3. Sélectionnez la connexion d'appairage de VPC en attente (statut pending-acceptance).
4. Choisissez Actions.
5. Dans la liste déroulante Actions, choisissez Accepter la demande.
6. Lorsque vous êtes invité à confirmer l'opération, choisissez Accepter la demande.
7. Choisissez Modifier mes tables de routage maintenant pour ajouter une route à la table de routage principale de votre VPC afin de pouvoir envoyer et recevoir du trafic via la connexion d'appairage.

Inspectez les tables de routage du VPC de la ressource. La route générée par Amazon VPC peut ne pas établir de connectivité, en fonction de la configuration du VPC de votre ressource. Vérifiez les conflits entre la nouvelle route et la configuration existante du VPC. Pour plus d'informations sur le dépannage, consultez Troubleshoot a VPC peering connection dans le Guide d'appairage de VPC Amazon Virtual Private Cloud.

Pour mettre à jour le groupe de sécurité de votre ressource

1. Ouvrez la https://console.aws.amazon.com/vpc/.
2. Dans le panneau de navigation, choisissez Groupes de sécurité.

3. Sélectionnez le groupe de sécurité pour votre ressource.
4. Choisissez Actions.
5. Dans la liste déroulante, choisissez Modifier les règles entrantes.
6. Choisissez Ajouter une règle.
7. Pour Source, saisissez l'ID de compte et de groupe de sécurité de votre fonction, séparés par une barre oblique (par exemple, 111122223333/sg-1a2b3c4d).
8. Choisissez Edit outbound rules (Modifier les règles sortantes).
9. Vérifiez si le trafic sortant est restreint. Les paramètres de VPC par défaut autorisent tout le trafic sortant. Si le trafic sortant est restreint, passez à l'étape suivante.
10. Choisissez Ajouter une règle.
11. Pour Destination, saisissez l'ID de compte et de groupe de sécurité de votre fonction, séparés par une barre oblique (par exemple, 111122223333/sg-1a2b3c4d).
12. Sélectionnez Enregistrer les règles.

Pour activer la résolution DNS pour votre connexion d'appairage

1. Ouvrez la https://console.aws.amazon.com/vpc/.
2. Dans le volet de navigation, choisissez Peering connections (Connexions d'appairage).
3. Sélectionnez votre connexion d'appairage.
4. Choisissez Actions.
5. Choisissez Modifier les paramètres DNS.
6. En-dessous de Résolution DNS acceptée, sélectionnez Autoriser le VPC demandeur à convertir le DNS des hôtes VPC accepteurs en IP privée.
7. Sélectionnez Enregistrer les modifications.

Mise à jour de la configuration VPC dans le compte de votre fonction

Connectez-vous au compte de votre fonction, puis mettez à jour la configuration du VPC.

Pour ajouter une route pour une connexion d'appairage de VPC

1. Ouvrez la https://console.aws.amazon.com/vpc/.
2. Dans le volet de navigation, choisissez Route tables (Tables de routage).

3. Cochez la case en regard du nom de la table de routage pour le sous-réseau que vous avez associé à votre fonction.
4. Choisissez Actions.
5. Choisissez Edit routes (Modifier des routes).
6. Choisissez Ajouter une route.
7. Pour Destination, saisissez le bloc CIDR du VPC de votre ressource.
8. Pour Cible, sélectionnez la connexion d'appairage de VPC.
9. Sélectionnez Enregistrer les modifications.

Pour de plus amples informations relatives aux éléments que vous pouvez rencontrer lors de la mise à jour de vos tables de routage, consultez Update your route tables for a VPC peering connection.

Pour mettre à jour le groupe de sécurité de votre fonction Lambda

1. Ouvrez la https://console.aws.amazon.com/vpc/.
2. Dans le panneau de navigation, choisissez Groupes de sécurité.
3. Choisissez Actions.
4. Choisissez Modifier les règles entrantes.
5. Choisissez Ajouter une règle.
6. Pour Source, saisissez l'ID de compte et de groupe de sécurité de votre ressource, séparés par une barre oblique (par exemple, 111122223333/sg-1a2b3c4d).
7. Sélectionnez Enregistrer les règles.

Pour activer la résolution DNS pour votre connexion d'appairage

1. Ouvrez la https://console.aws.amazon.com/vpc/.
2. Dans le volet de navigation, choisissez Peering connections (Connexions d'appairage).
3. Sélectionnez votre connexion d'appairage.
4. Choisissez Actions.
5. Choisissez Modifier les paramètres DNS.
6. Sous Résolution DNS du demandeur, sélectionnez Autoriser le VPC accepteur à convertir le DNS des hôtes VPC demandeur en IP privée.
7. Sélectionnez Enregistrer les modifications.

Test de votre fonction

Pour créer un événement de test et inspecter la sortie de votre fonction

1. Dans le volet Source du code, choisissez Test.
2. Sélectionnez Créer un événement.
3. Dans le panneau JSON d’événement, remplacez les valeurs par défaut par une entrée adaptée à votre fonction Lambda.
4. Sélectionnez Invoquer .
5. Dans l’onglet Résultats de l’exécution, vérifiez que Response contient le résultat attendu.

En outre, vous pouvez consulter les journaux de votre fonction pour vérifier qu’ils correspondent à vos attentes.

Pour consulter les enregistrements d'invocation de votre fonction dans Logs CloudWatch

1. Choisissez l’onglet Surveiller.
2. Choisissez Afficher CloudWatch les journaux.
3. Dans l'onglet Flux de journaux, choisissez le flux de journaux pour l'invocation de votre fonction.
4. Vérifiez que vos journaux sont conformes à vos attentes.

Activation de l’accès Internet pour les fonctions Lambda connectées à un VPC

Par défaut, les fonctions Lambda s’exécutent dans un VPC géré par Lambda disposant d’un accès à Internet. Pour accéder aux ressources d’un VPC dans votre compte, vous pouvez ajouter une configuration VPC à une fonction. Cela limite la fonction aux ressources de ce VPC, sauf si le VPC dispose d’un accès à Internet. Cette page explique comment fournir un accès Internet aux fonctions Lambda connectées au VPC.

Je n’ai pas encore de VPC

Créer le VPC

Le flux de travail de création de VPC crée toutes les ressources VPC requises pour qu’une fonction Lambda accède à l’Internet public à partir d’un sous-réseau privé, y compris les sous-réseaux, la passerelle NAT, la passerelle Internet et les entrées de table de routage.

Pour créer le VPC

1. Ouvrez la console Amazon VPC à l'adresse https://console.aws.amazon.com/vpc/.
2. Sur le tableau de bord, choisissez Créer un VPC.
3. Sous Ressources à créer, choisissez VPC et plus encore.
4. Configurer le VPC
    a. Pour Name tag auto-generation (Génération automatique de balises de nom), saisissez un nom pour le VPC.
    b. Pour Bloc d'adresse CIDR IPv4, vous pouvez conserver la suggestion par défaut ou saisir le bloc d'adresse CIDR requis par votre application ou votre réseau.
    c. Si votre application communique à l'aide d'adresses IPv6, choisissez Bloc d'adresse CIDR IPv6, Bloc d'adresse CIDR IPv6 fourni par Amazon.
5. Configurez les sous-réseaux
    a. Pour Nombre de zones de disponibilité, choisissez 2. Nous recommandons au moins deux zones de disponibilité pour une haute disponibilité.
    b. Pour Number of public subnets (Nombre de sous-réseaux publics), choisissez 2.
    c. Pour Number of private subnets (Nombre de sous-réseaux privés), choisissez 2.

d. Vous pouvez conserver le bloc d'adresse CIDR par défaut pour le sous-réseau public ou développer Personnaliser les blocs d'adresse CIDR du sous-réseau et saisir un bloc d'adresse CIDR. Pour plus d'informations, consultez Subnet CIDR blocks.

6. Pour Passerelles NAT, choisissez 1 par zone de disponibilité afin d'améliorer la résilience.
7. Pour Passerelle Internet de sortie uniquement, choisissez Oui si vous avez choisi d'inclure un bloc d'adresse CIDR IPv6.
8. Pour Points de terminaison de VPC, conservez la valeur par défaut (Passerelle S3). Cette option n'entraîne pas de frais. Pour plus d'informations, consultez Types of VPC endpoints for Amazon S3.
9. Pour Options DNS, conservez les paramètres par défaut.
10. Sélectionnez Create VPC (Créer un VPC).

Configurer la fonction Lambda

Pour configurer un VPC lorsque vous créez une fonction

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez Créer une fonction.
3. Sous Informations de base, dans Nom de fonction, entrez un nom pour votre fonction.
4. Développez Advanced settings (Paramètres avancés).
5. Sélectionnez Activer le VPC, puis choisissez un VPC.
6. (Facultatif) Pour autoriser le trafic IPv6 sortant, sélectionnez Autoriser le trafic IPv6 pour les sous-réseaux à double pile.
7. Pour Sous-réseaux, sélectionnez tous les sous-réseaux privés. Les sous-réseaux privés peuvent accéder à Internet via une passerelle NAT. La connexion d'une fonction à un sous-réseau public ne lui donne pas accès à Internet.

> **Note**
>
> Si vous avez sélectionné Autoriser le trafic IPv6 pour les sous-réseaux à double pile, tous les sous-réseaux sélectionnés doivent avoir un bloc CIDR IPv4 et un bloc CIDR IPv6.

8. Pour les groupes de sécurité, sélectionnez un groupe de sécurité qui autorise le trafic sortant.

9. Choisissez Créer une fonction.

Lambda crée automatiquement un rôle d'exécution avec la politique AWSLambdaVPCAccessExecutionRole gérée par AWS. Les autorisations de cette politique ne sont nécessaires que pour créer des interfaces réseau élastiques pour la configuration VPC, et non pour invoquer votre fonction. Pour appliquer les autorisations de moindre privilège, vous pouvez supprimer la politique AWSLambdaVPCAccessExecutionRole de votre rôle d'exécution après avoir créé la fonction et la configuration VPC. Pour de plus amples informations, consultez Autorisations IAM requises.

Pour configurer un VPC pour une fonction existante

Pour ajouter une configuration VPC à une fonction existante, le rôle d'exécution de la fonction doit être autorisé à créer et à gérer des interfaces réseau élastiques. Ces autorisations sont incluses dans la politique AWSLambdaVPCAccessExecutionRole gérée AWS. Pour appliquer les autorisations de moindre privilège, vous pouvez supprimer la politique AWSLambdaVPCAccessExecutionRole de votre rôle d'exécution après avoir créé la configuration VPC.

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Choisissez l'onglet Configuration puis choisissez VPC.
4. Sous VPC, choisissez Modifier.
5. Sélectionnez le VPC.
6. (Facultatif) Pour autoriser le trafic IPv6 sortant, sélectionnez Autoriser le trafic IPv6 pour les sous-réseaux à double pile.
7. Pour Sous-réseaux, sélectionnez tous les sous-réseaux privés. Les sous-réseaux privés peuvent accéder à Internet via une passerelle NAT. La connexion d'une fonction à un sous-réseau public ne lui donne pas accès à Internet.

   > **Note**
   >
   > Si vous avez sélectionné Autoriser le trafic IPv6 pour les sous-réseaux à double pile, tous les sous-réseaux sélectionnés doivent avoir un bloc CIDR IPv4 et un bloc CIDR IPv6.

8. Pour les groupes de sécurité, sélectionnez un groupe de sécurité qui autorise le trafic sortant.

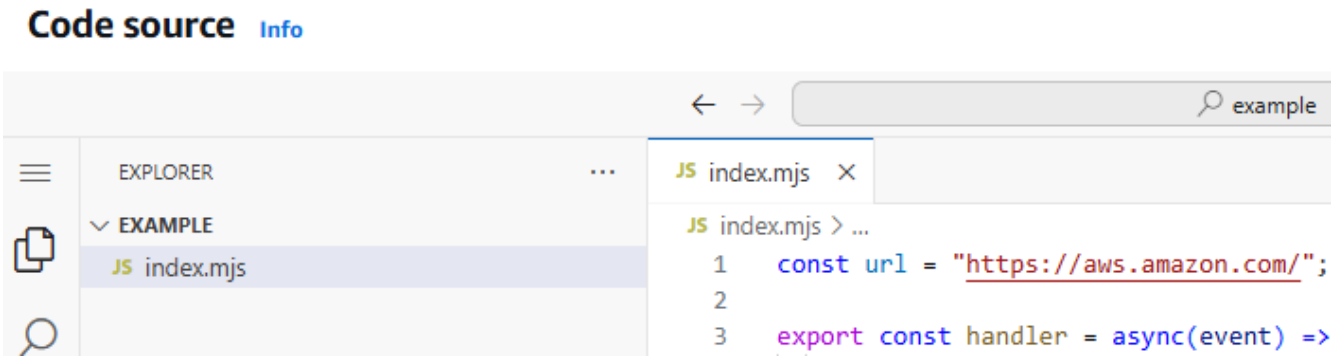
9. Choisissez Enregistrer.

Tester la fonction

Utilisez l'exemple de code suivant pour vérifier que votre fonction connectée au VPC peut accéder à l'Internet public. En cas de succès, le code renvoie un code d'état `200`. En cas d'échec, la fonction expire.

Node.js

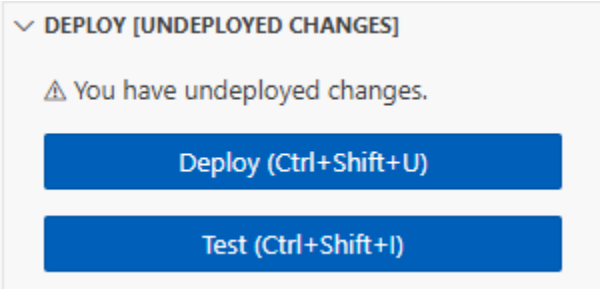
1. Dans le volet Source du code de la console Lambda, collez le code suivant dans le fichier index.mjs. La fonction envoie une requête HTTP GET à un point de terminaison public et renvoie le code de réponse HTTP pour tester si la fonction a accès à l'Internet public.



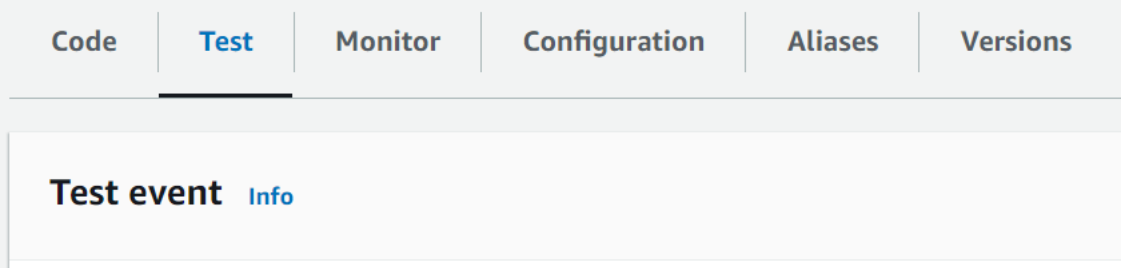
   Example – Requête HTTP avec async/await

```
const url = "https://aws.amazon.com/";

export const handler = async(event) => {
    try {
        const res = await fetch(url);
        console.info("status", res.status);
        return res.status;
    }
    catch (e) {
        console.error(e);
        return 500;
    }
};
```

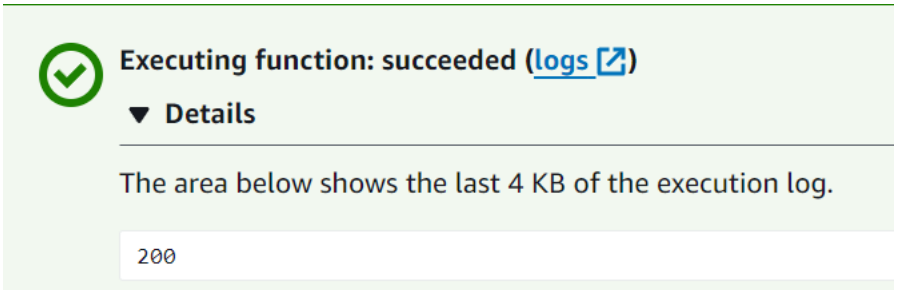
2. Dans la section DÉPLOYER, choisissez Déployer pour mettre à jour le code de votre fonction :



3. Choisissez l'onglet Test.



4. Sélectionnez Tester).
5. La fonction renvoie un code d'état `200`. Cela signifie que la fonction dispose d'un accès Internet sortant.



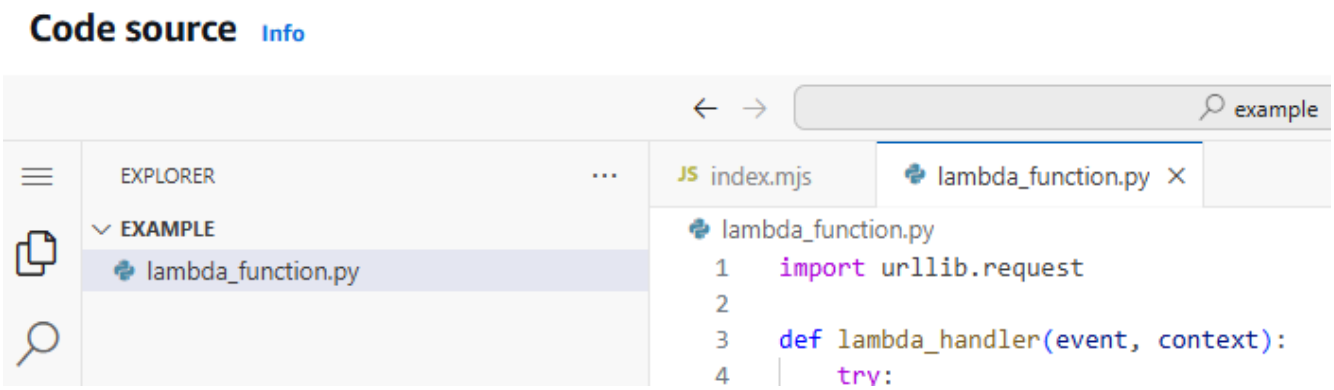
Si la fonction ne parvient pas à accéder à l'Internet public, un message d'erreur de ce type s'affiche :

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

Python

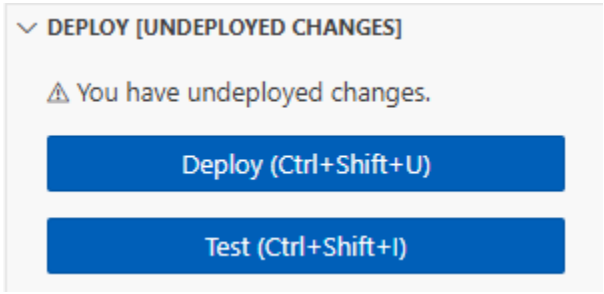
1. Dans le volet Source du code de la console Lambda, collez le code suivant dans le fichier lambda_function.py. La fonction envoie une requête HTTP GET à un point de terminaison

public et renvoie le code de réponse HTTP pour tester si la fonction a accès à l'Internet public.

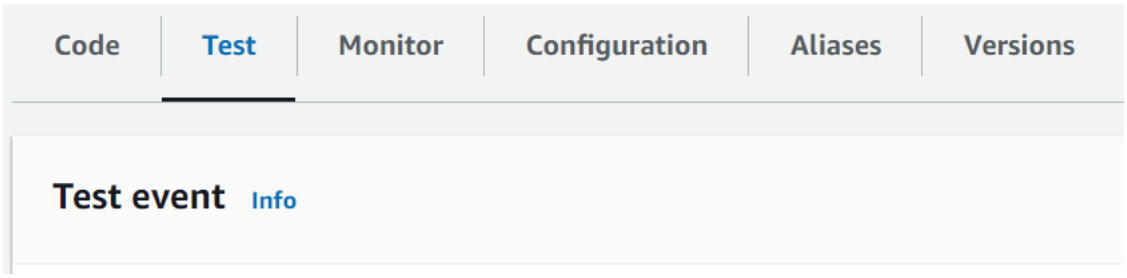



```
import urllib.request

def lambda_handler(event, context):
    try:
        response = urllib.request.urlopen('https://aws.amazon.com')
        status_code = response.getcode()
        print('Response Code:', status_code)
        return status_code
    except Exception as e:
        print('Error:', e)
        raise e
```

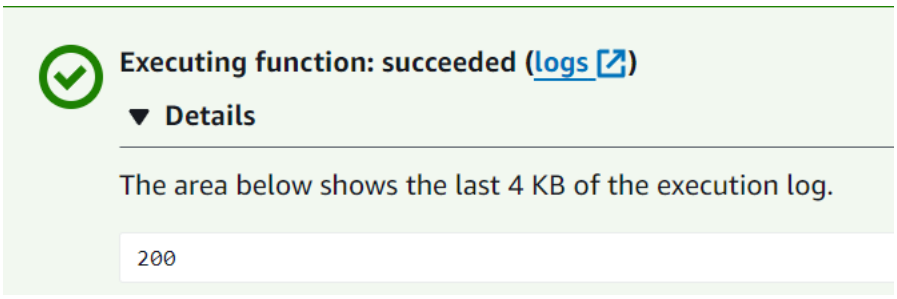
2. Dans la section DÉPLOYER, choisissez Déployer pour mettre à jour le code de votre fonction :



3. Choisissez l'onglet Test.



4. Sélectionnez Tester).
5. La fonction renvoie un code d'état `200`. Cela signifie que la fonction dispose d'un accès Internet sortant.



Si la fonction ne parvient pas à accéder à l'Internet public, un message d'erreur de ce type s'affiche :

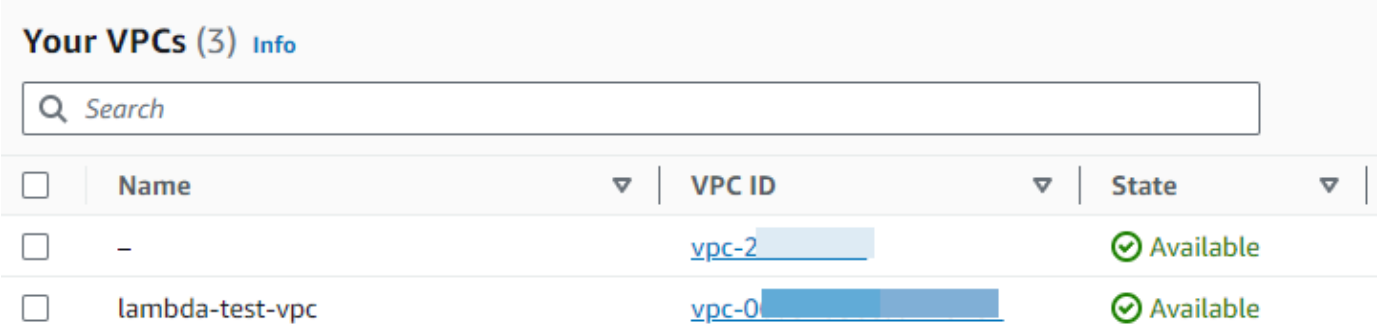
```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

J'ai déjà un VPC

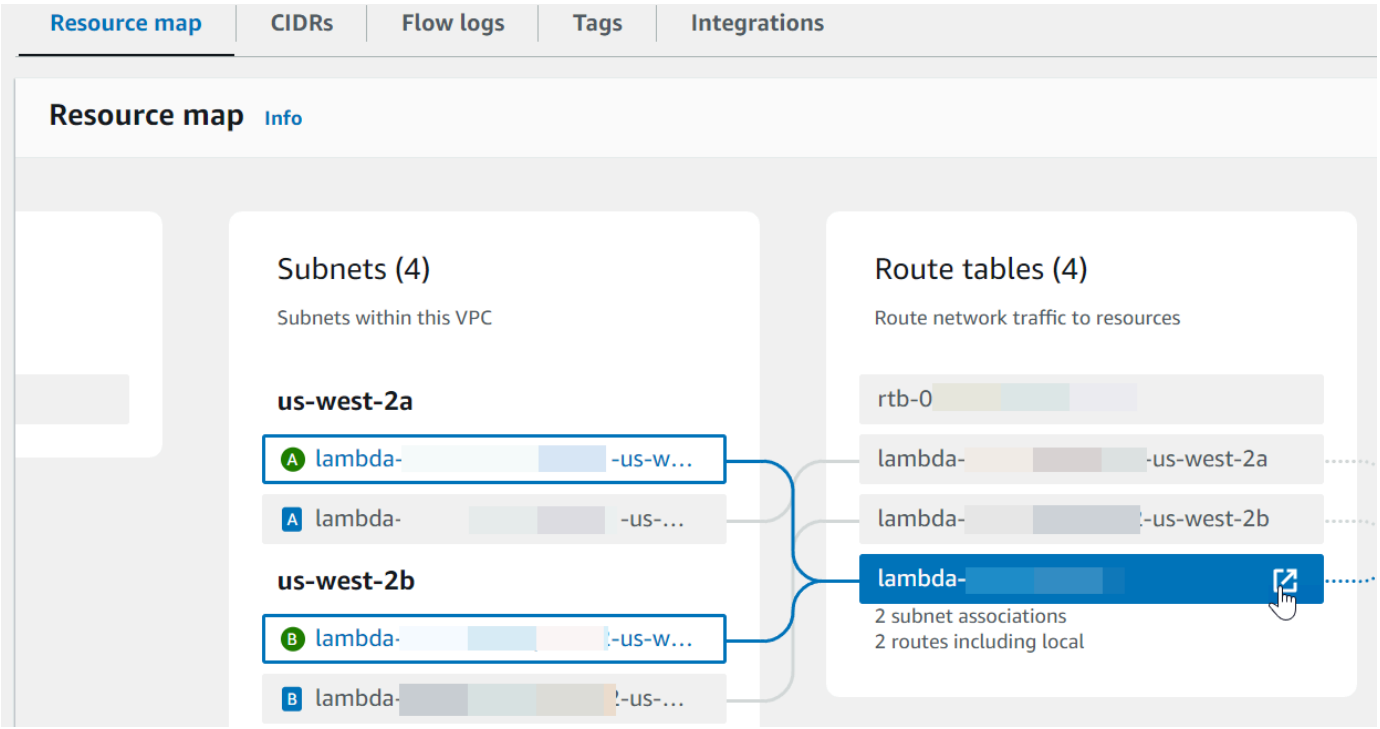
Si vous possédez déjà un VPC mais que vous devez configurer un accès Internet public pour une fonction Lambda, procédez comme suit. Cette procédure suppose que votre VPC possède au moins deux sous-réseaux. Si vous n'avez pas deux sous-réseaux, consultez Create a subnet dans le Guide de l'utilisateur Amazon VPC.

Vérification de la configuration de la table de routage

1. Ouvrez la console Amazon VPC à l'adresse https://console.aws.amazon.com/vpc/.
2. Choisissez l'ID de VPC.



3. Faites défiler la page jusqu'à la section Carte des ressources. Notez les mappages des tables de routage. Ouvrez chaque table de routage mappée à un sous-réseau.

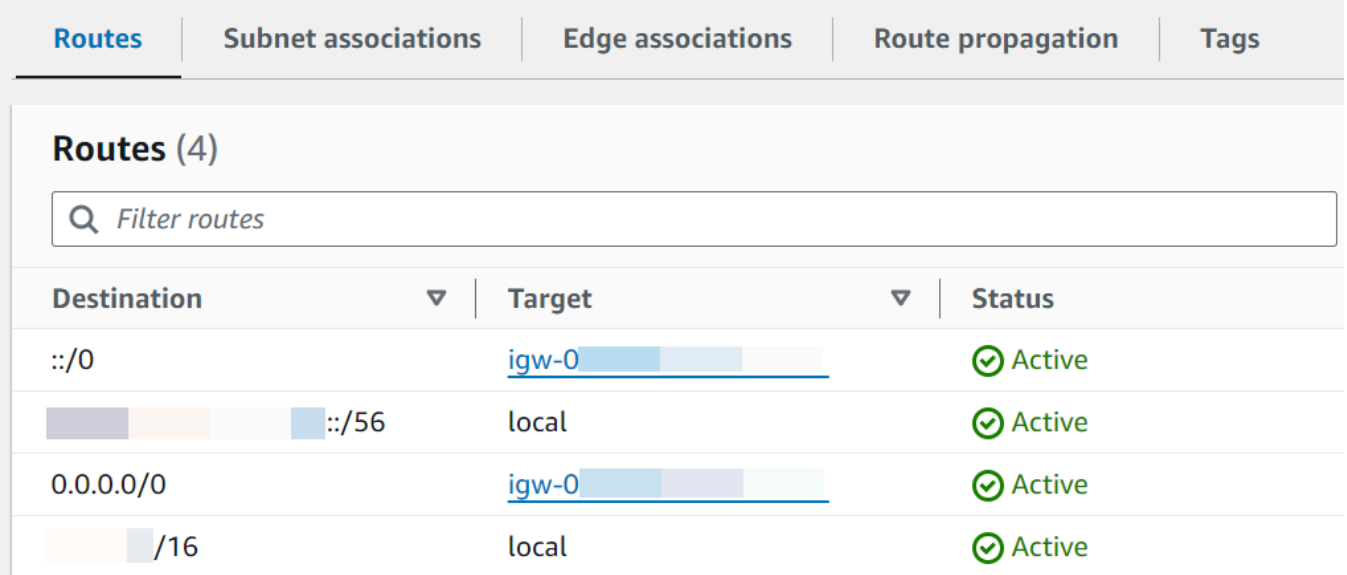


4. Faites défiler jusqu'à l'onglet Routes. Passez en revue les itinéraires pour déterminer si votre VPC possède les deux tables de routage suivantes. Chacune de ces exigences doit être satisfaite par une table de routage distincte.

   - Le trafic Internet (`0.0.0.0/0` pour IPv4, `::/0` pour IPv6) est acheminé vers une passerelle Internet (`igw-xxxxxxxxxx`). Cela signifie que le sous-réseau associé à la table de routage est un sous-réseau public.

> **Note**
>
> Si votre sous-réseau ne possède pas de bloc d'adresse CIDR IPv6, vous ne verrez que la route IPv4 (`0.0.0.0/0`).

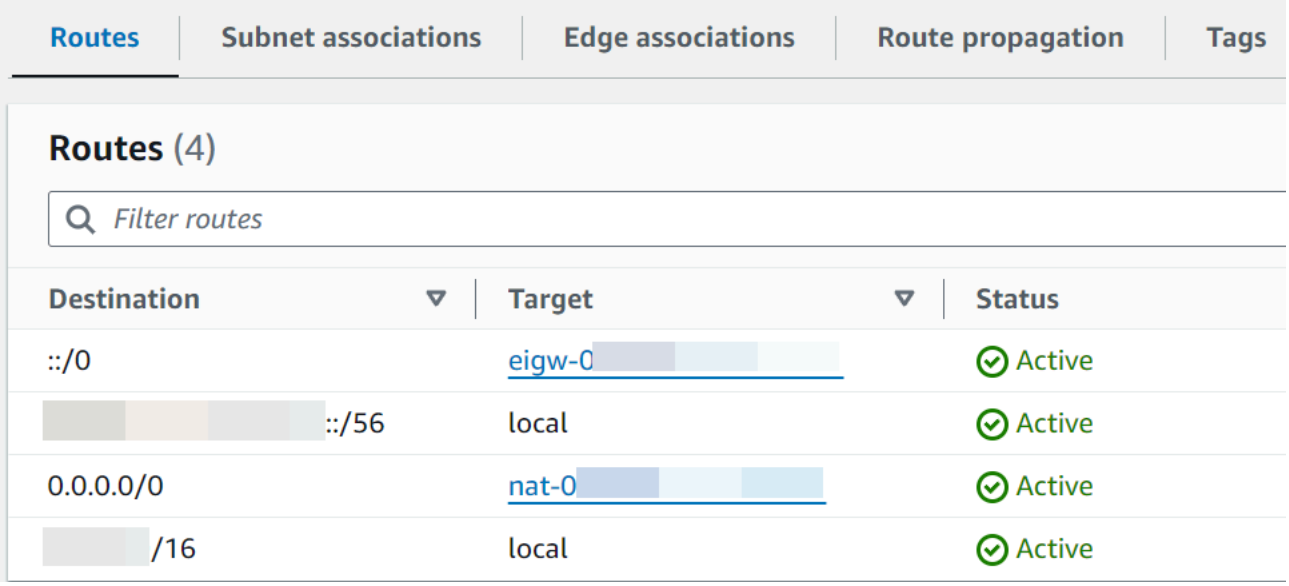
Example Table de routage de sous-réseau public



- Le trafic Internet pour IPv4 (`0.0.0.0/0`) est acheminé vers une passerelle NAT (`nat-xxxxxxxxxx`) associée à un sous-réseau public. Cela signifie que le sous-réseau est un sous-réseau privé qui peut accéder à l'Internet via la passerelle NAT.

> **Note**
>
> Si votre sous-réseau comporte un bloc CIDR IPv6, la table de routage doit également acheminer le trafic Internet IPv6 (`::/0`) vers une passerelle Internet de sortie uniquement (`eigw-xxxxxxxxxx`). Si votre sous-réseau ne possède pas de bloc d'adresse CIDR IPv6, vous ne verrez que la route IPv4 (`0.0.0.0/0`).

Example table de routage de sous-réseau privé



5. Répétez l'étape précédente jusqu'à ce que vous ayez examiné chaque table de routage associée à un sous-réseau de votre VPC et confirmé que vous disposez d'une table de routage avec une passerelle Internet et d'une table de routage avec une passerelle NAT.

   Si vous n'avez pas deux tables de routage, l'une avec une route vers une passerelle Internet et l'autre avec une route vers une passerelle NAT, suivez ces étapes pour créer les ressources manquantes et les entrées de table de routage.

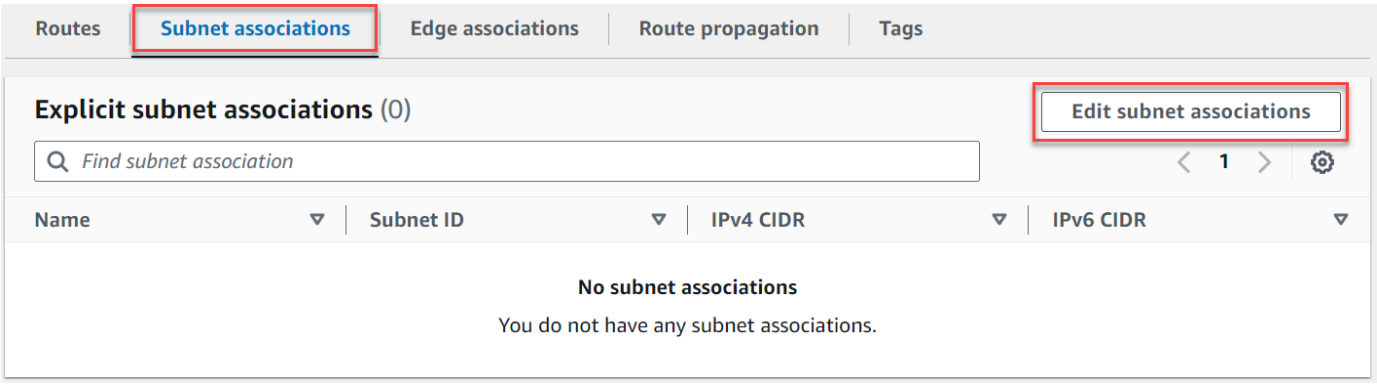
Création d'une table de routage

Suivez les étapes ci-dessous pour créer une table de routage et l'associer à un sous-réseau.

Pour créer une table de routage personnalisée à l'aide de la console Amazon VPC

1. Ouvrez la console Amazon VPC à l'adresse https://console.aws.amazon.com/vpc/.
2. Dans le volet de navigation, choisissez Route tables (Tables de routage).
3. Choisissez Créer une table de routage.
4. (Facultatif) Pour Nom, entrez un nom pour votre table de routage.
5. Pour VPC, choisissez votre VPC.
6. (Facultatif) Pour ajouter une identification, choisissez Add new tag (Ajouter une identification) et saisissez la clé et la valeur de l'identification.

7. Choisissez Créer une table de routage.
8. Sur l'onglet Associations de sous-réseau, choisissez Modifier les associations de sous-réseau.



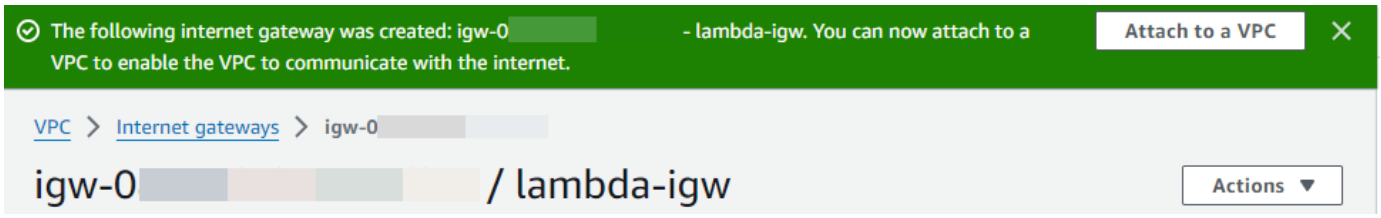
9. Sélectionnez la case à cocher pour le sous-réseau à associer à la table de routage.
10. Choisissez Save associations (Enregistrer les associations).

Création d'une passerelle Internet

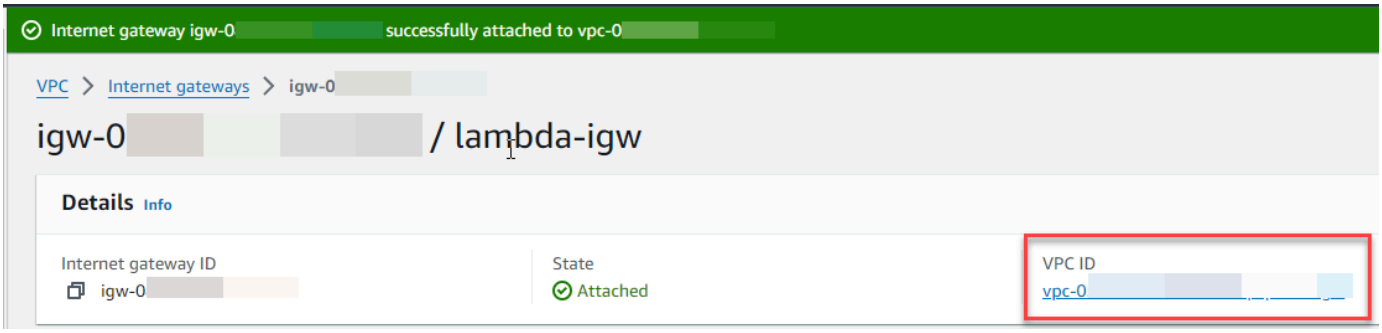
Procédez comme suit pour créer une passerelle Internet, l'associer à votre VPC et l'ajouter à la table de routage de votre sous-réseau public.

Pour créer une passerelle Internet

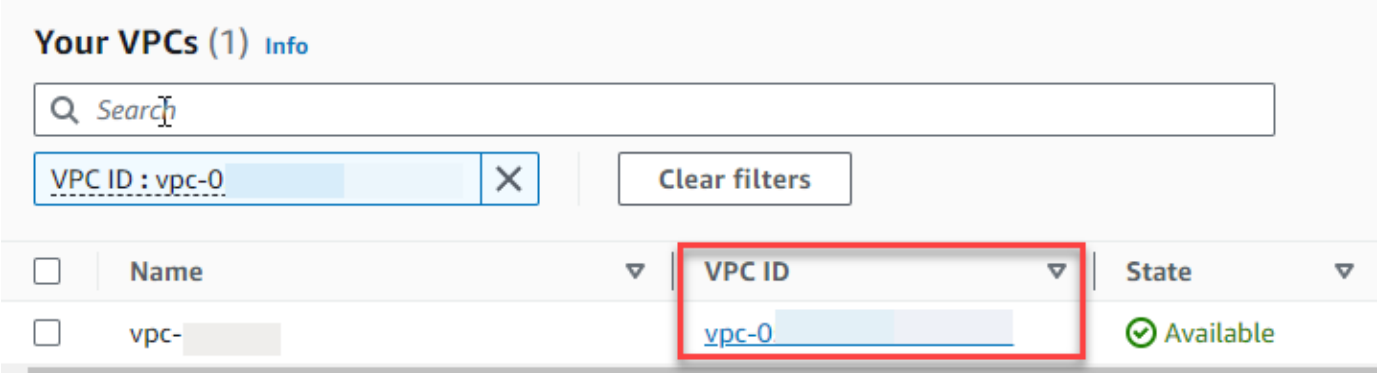
1. Ouvrez la console Amazon VPC à l'adresse https://console.aws.amazon.com/vpc/.
2. Dans le panneau de navigation, choisissez Passerelles Internet.
3. Choisissez Créer une passerelle Internet.
4. (Facultatif) Saisissez un nom pour votre passerelle Internet.
5. (Facultatif) Pour ajouter une identification, choisissez Add new tag (Ajouter une identification) et saisissez la clé et la valeur de l'identification.
6. Choisissez Créer une passerelle Internet.
7. Choisissez Attacher à un VPC dans la bannière en haut de l'écran, sélectionnez un VPC disponible, puis choisissez Attacher une passerelle Internet.



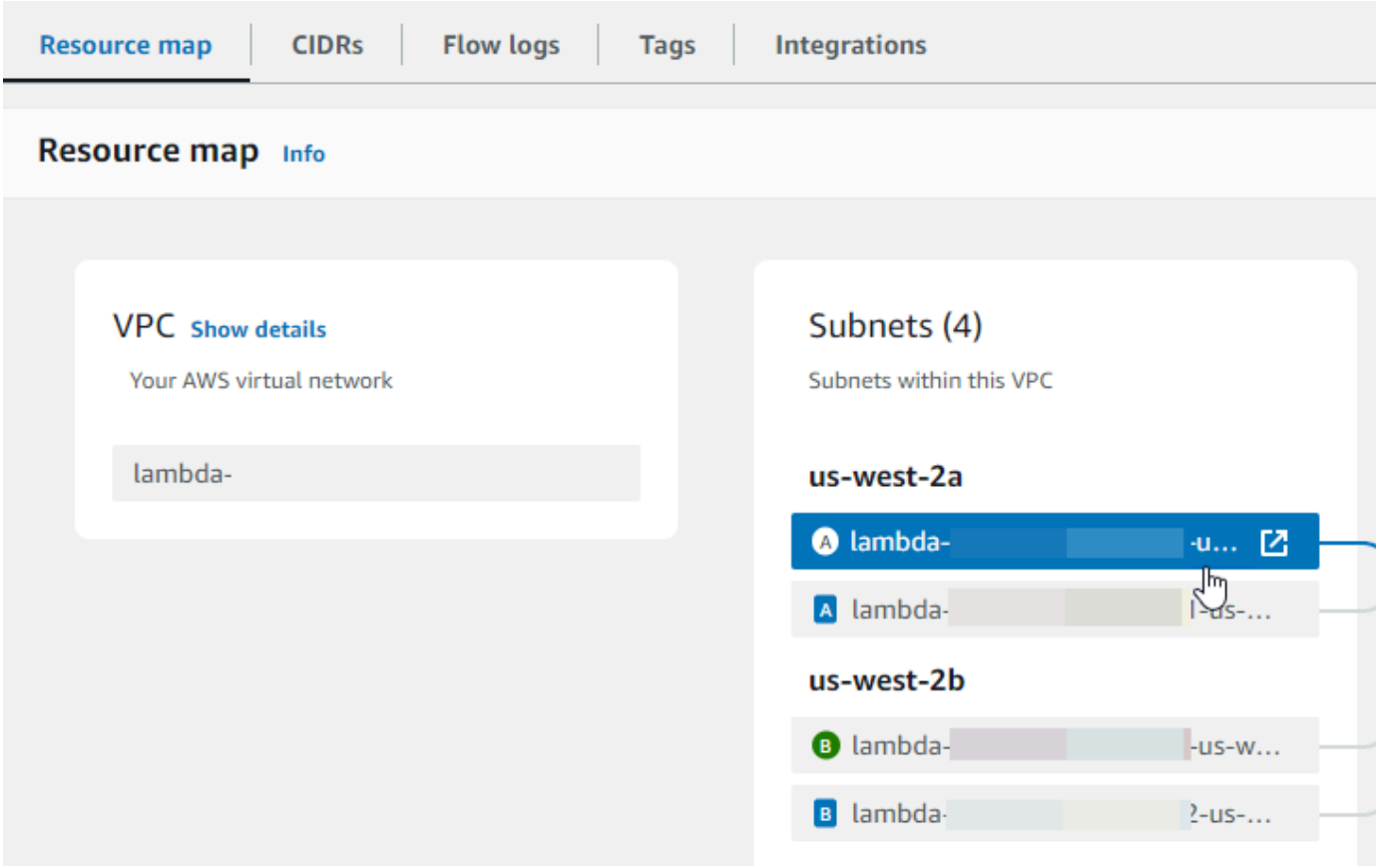
8. Choisissez l'ID de VPC.



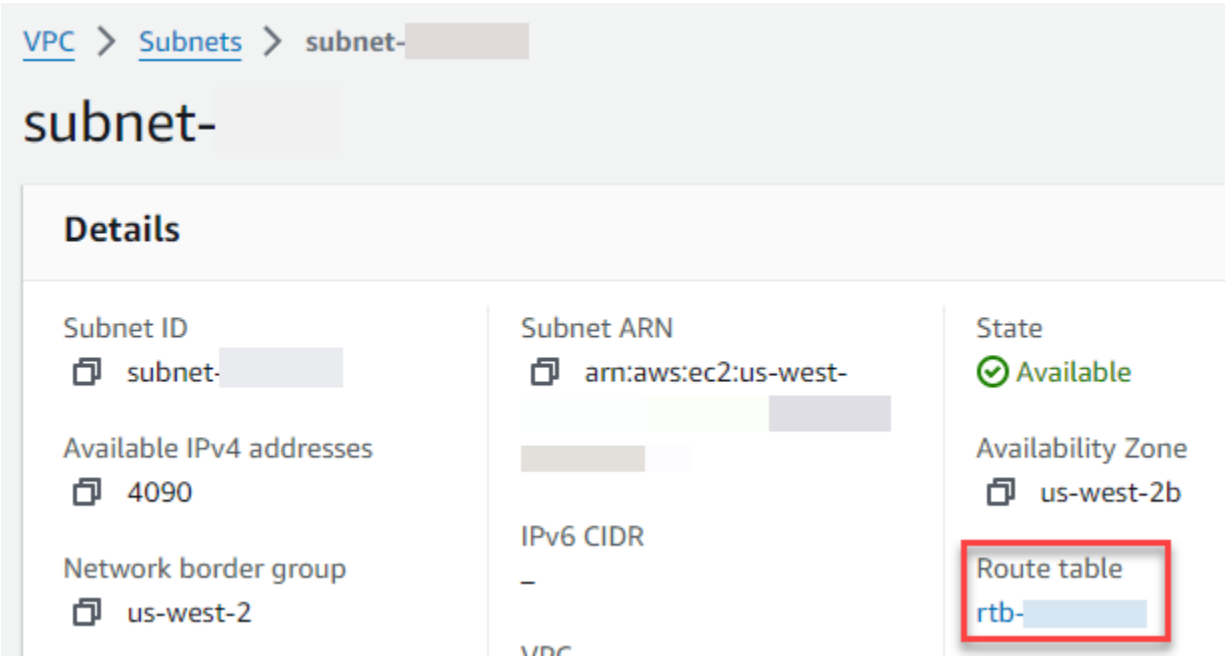
9. Sélectionnez à nouveau l'ID de VPC pour ouvrir la page des détails.



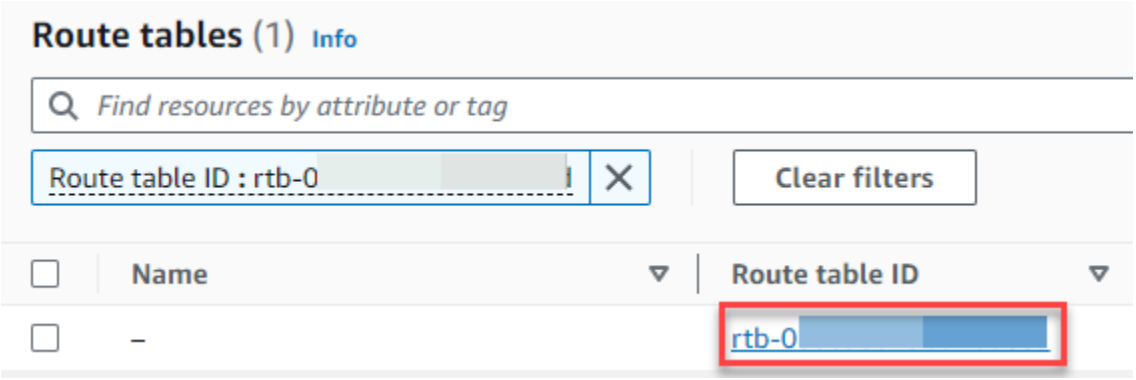
10. Faites défiler vers le bas jusqu'à la section Carte des ressources et choisissez un sous-réseau. Les détails du sous-réseau s'affichent dans un nouvel onglet.



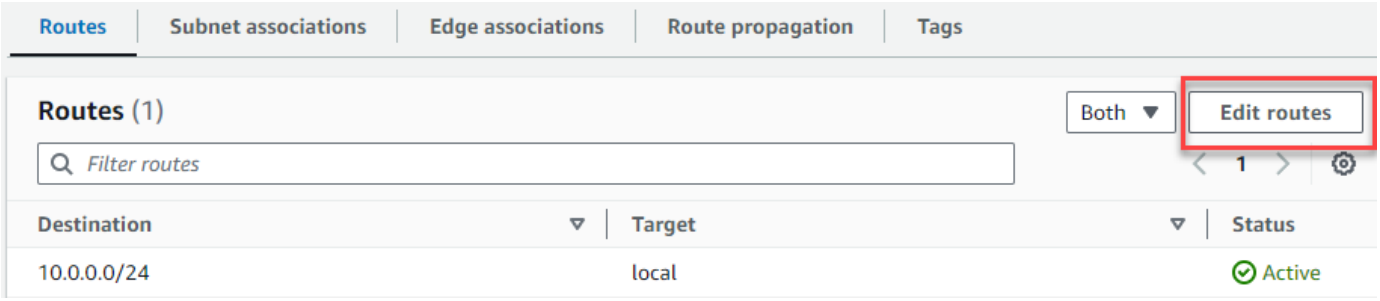
11. Cliquez sur le lien sous Table de routage.



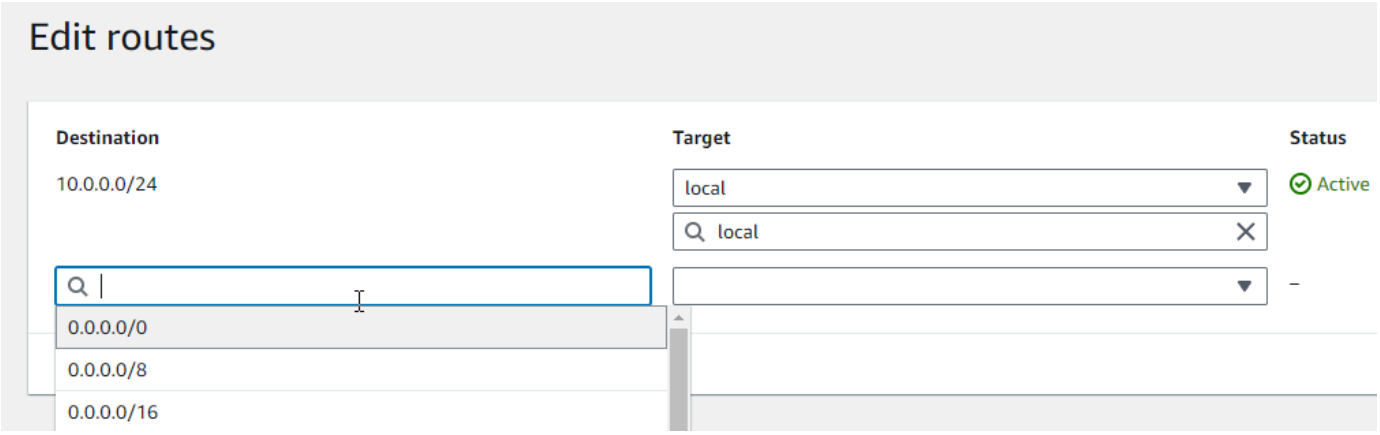
12. Choisissez l'ID de la table de routage pour ouvrir la page de détails de la table de routage.



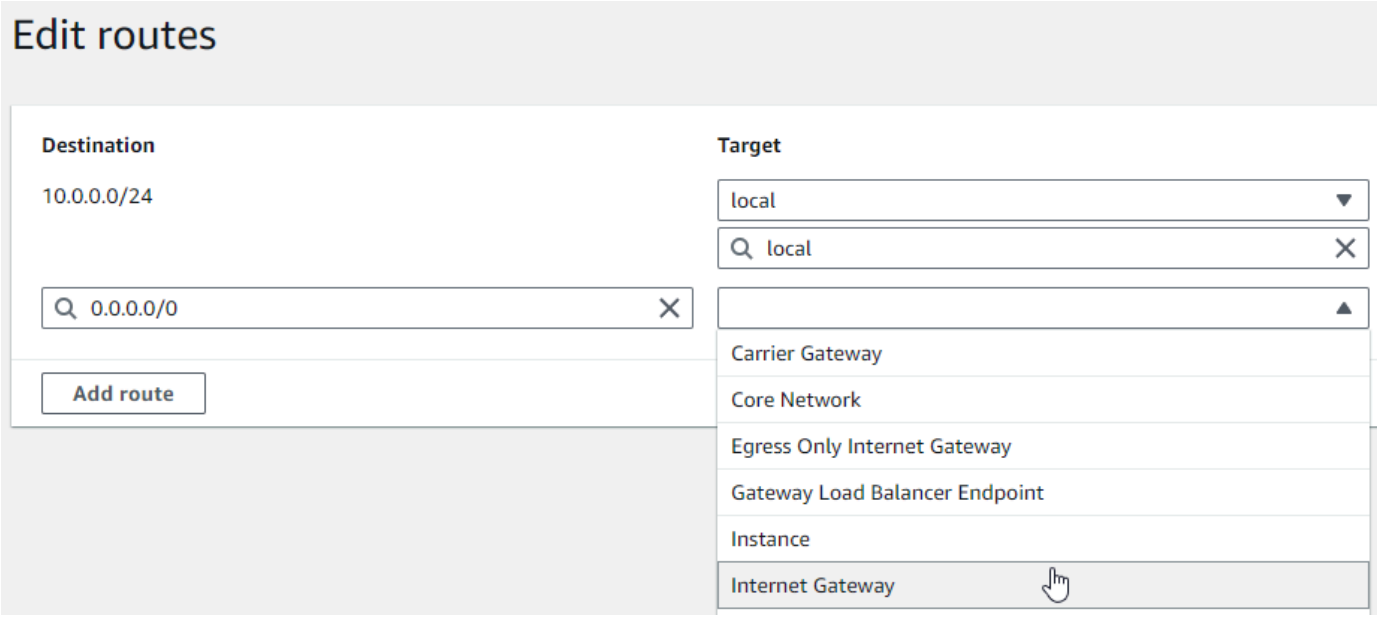
13. Sous Routes, choisissez la Modifier des routes.



14. Choisissez Ajouter une route, puis saisissez `0.0.0.0/0` dans la zone Destination.



15. Pour Cible, sélectionnez Passerelle Internet, puis choisissez la passerelle Internet que vous avez créée précédemment. Si votre sous-réseau comporte un bloc CIDR IPv6, vous devez également ajouter une route pour `::/0` vers la même passerelle Internet.



16. Sélectionnez Enregistrer les modifications.

Créer une passerelle NAT

Suivez ces étapes pour créer une passerelle NAT, l'associer à un sous-réseau public, puis l'ajouter à la table de routage de votre sous-réseau privé.

Pour créer une passerelle NAT et l'associer à un sous-réseau public

1. Dans le volet de navigation, sélectionnez Passerelles NAT.
2. Sélectionnez Créer une passerelle NAT.
3. (Facultatif) Saisissez un nom pour votre passerelle NAT.
4. Pour Sous-réseau, sélectionnez le sous-réseau public dans votre VPC. (Un sous-réseau public est un sous-réseau qui a une entrée de table de routage comportant une route vers une passerelle Internet)

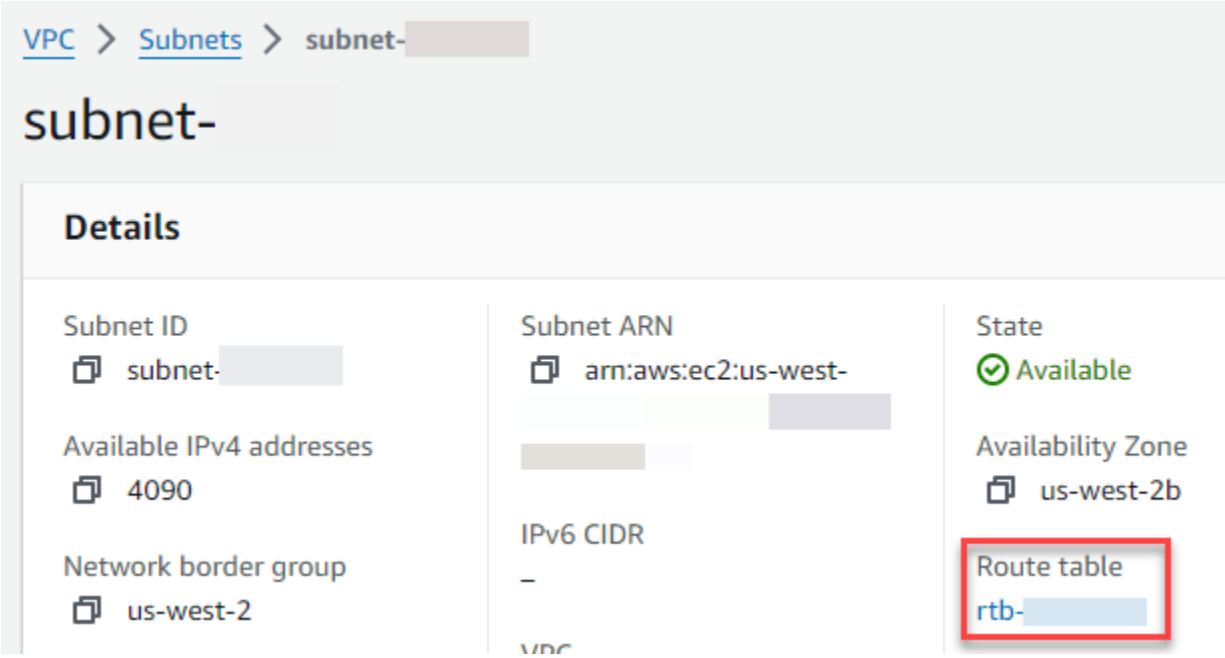
> **Note**
>
> Les passerelles NAT sont associées à un sous-réseau public, mais l'entrée de la table de routage se trouve dans le sous-réseau privé.

5. Pour ID d'allocation IP élastique, sélectionnez une adresse IP élastique ou choisissez Allouer une adresse IP élastique.

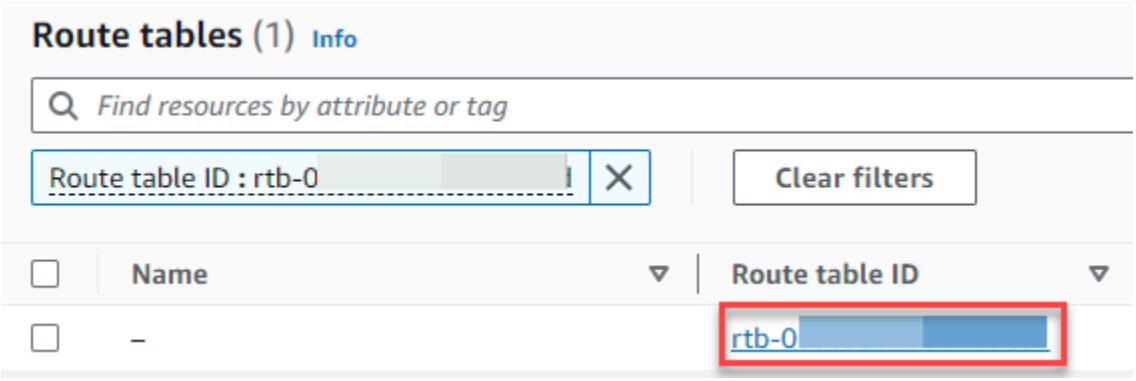
6. Sélectionnez Créer une passerelle NAT.

Pour ajouter une route vers la passerelle NAT dans la table de routage du sous-réseau privé

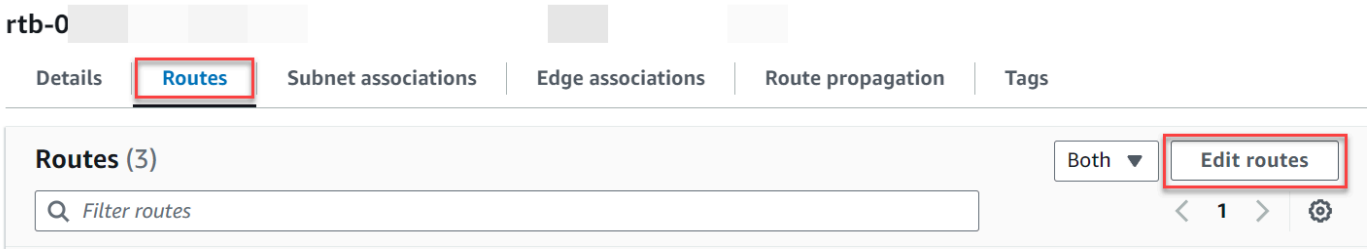
1. Dans le panneau de navigation, choisissez Subnets (Sous-réseaux).
2. Sélectionnez un sous-réseau privé dans votre VPC. (Un sous-réseau privé est un sous-réseau qui ne comporte pas de route vers une passerelle Internet dans sa table de routage)
3. Cliquez sur le lien sous Table de routage.



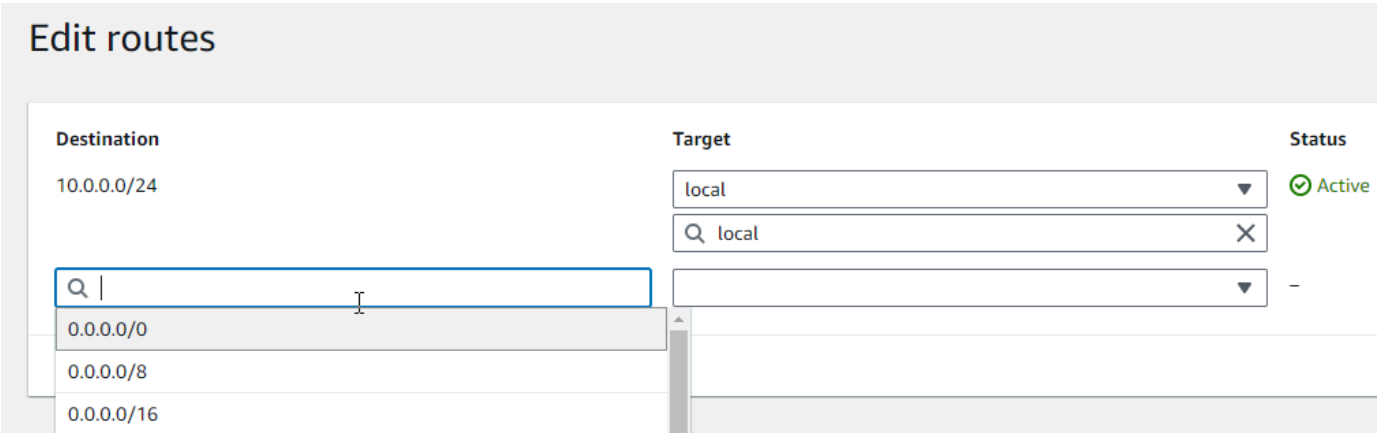
4. Choisissez l'ID de la table de routage pour ouvrir la page de détails de la table de routage.



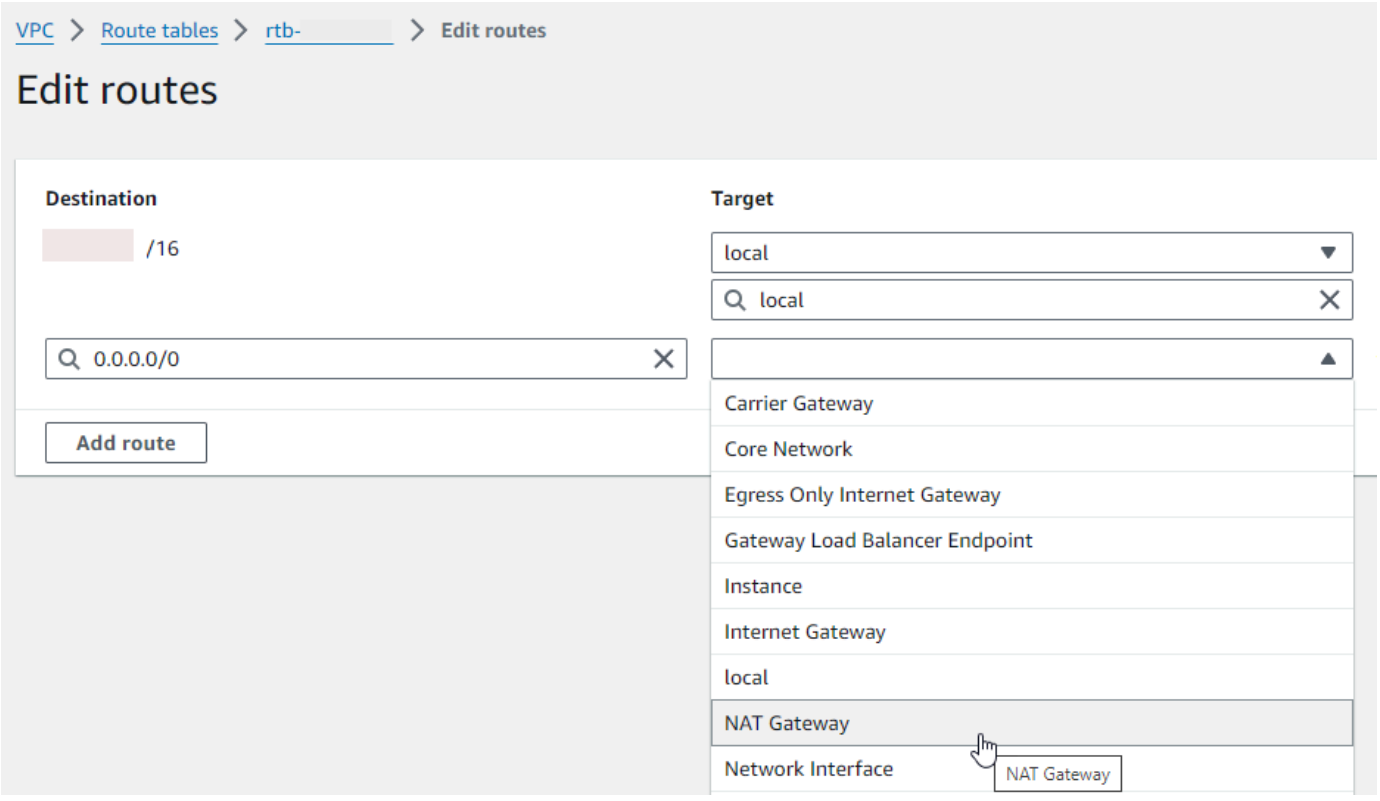
5. Choisissez l'onglet Routes, puis Modifier des routes.



6. Choisissez Ajouter une route, puis saisissez `0.0.0.0/0` dans la zone Destination.



7. Pour Cible, choisissez Passerelle NAT, puis sélectionnez la passerelle NAT que vous avez créée.



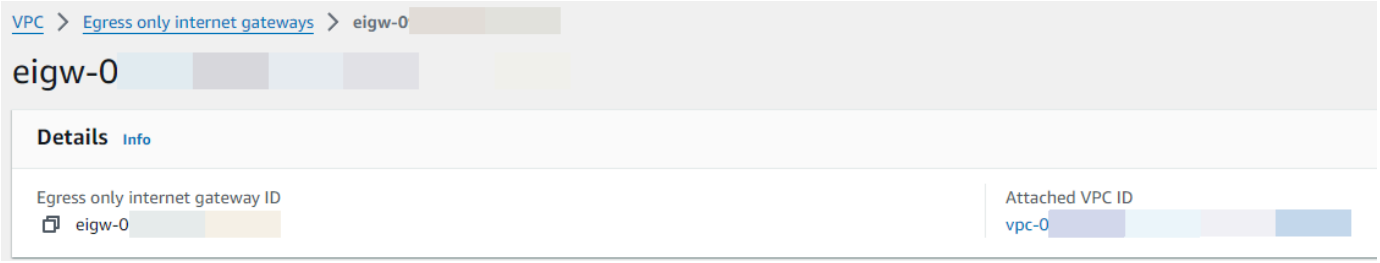
8. Sélectionnez Enregistrer les modifications.

Création d'une passerelle Internet de sortie uniquement (limité à IPv6)

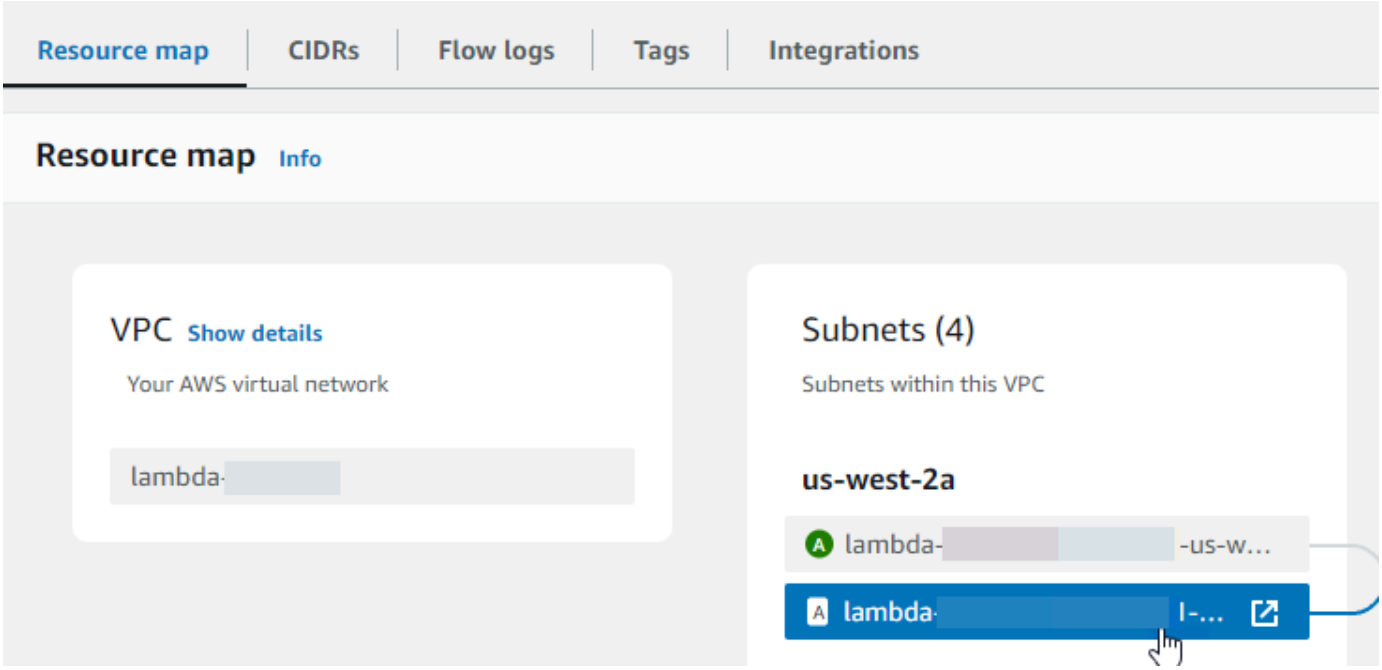
Procédez comme suit pour créer une passerelle Internet de sortie uniquement et l'ajouter à la table de routage de votre sous-réseau privé.

Pour créer une passerelle Internet de sortie uniquement

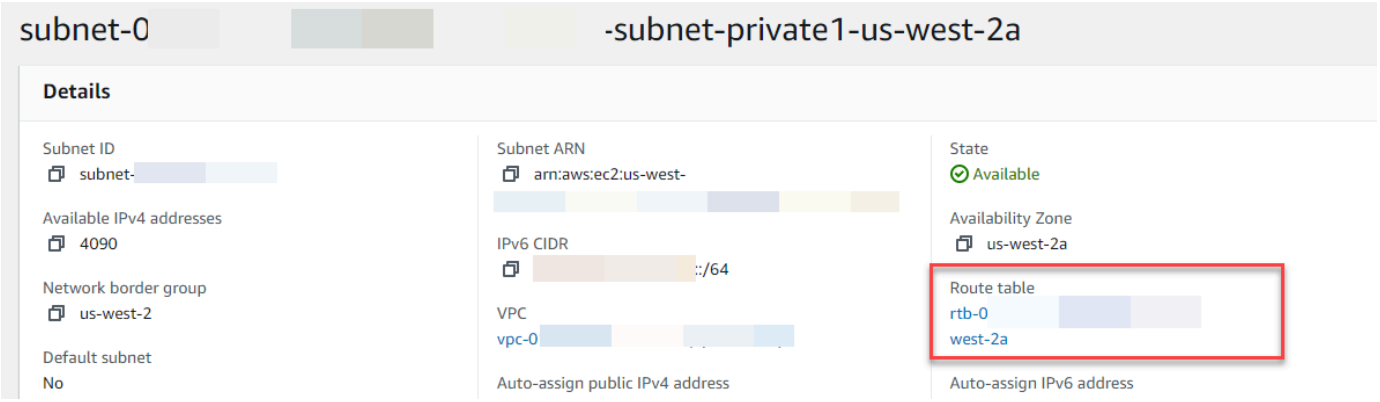
1. Dans le volet de navigation, choisissez Passerelles Internet de sortie uniquement.
2. Choisissez Créer une passerelle Internet de sortie uniquement.
3. (Facultatif) Saisissez un nom.
4. Sélectionnez le VPC dans lequel créer la passerelle Internet de sortie uniquement.
5. Choisissez Créer une passerelle Internet de sortie uniquement.
6. Choisissez le lien sous ID du VPC attaché.



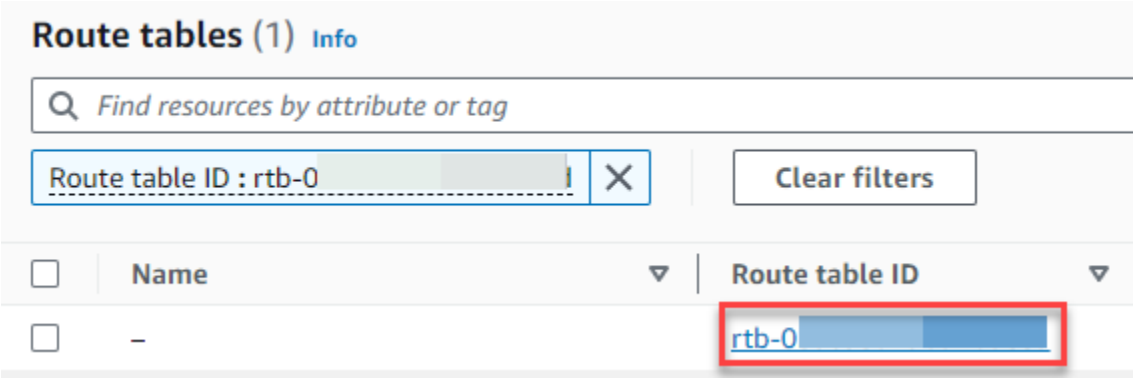
7. Sélectionnez le lien sous l'ID de VPC pour ouvrir la page des détails.
8. Faites défiler vers le bas jusqu'à la section Carte des ressources et choisissez un sous-réseau privé. (Un sous-réseau privé est un sous-réseau qui ne comporte pas de route vers une passerelle Internet dans sa table de routage) Les détails du sous-réseau s'affichent dans un nouvel onglet.



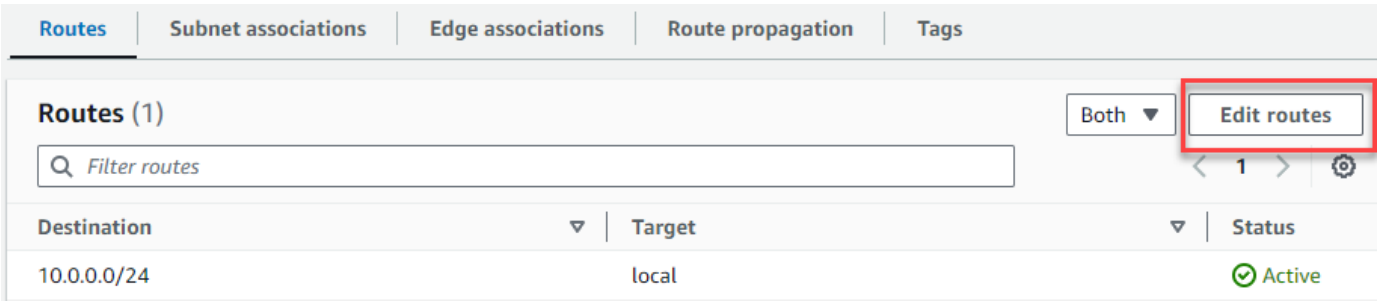
9. Cliquez sur le lien sous Table de routage.



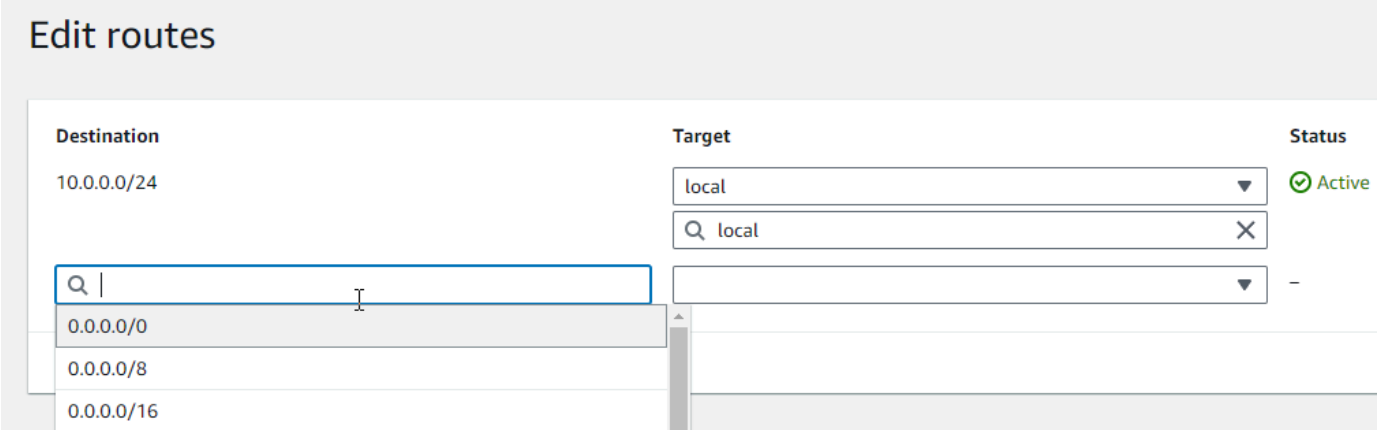
10. Choisissez l'ID de la table de routage pour ouvrir la page de détails de la table de routage.



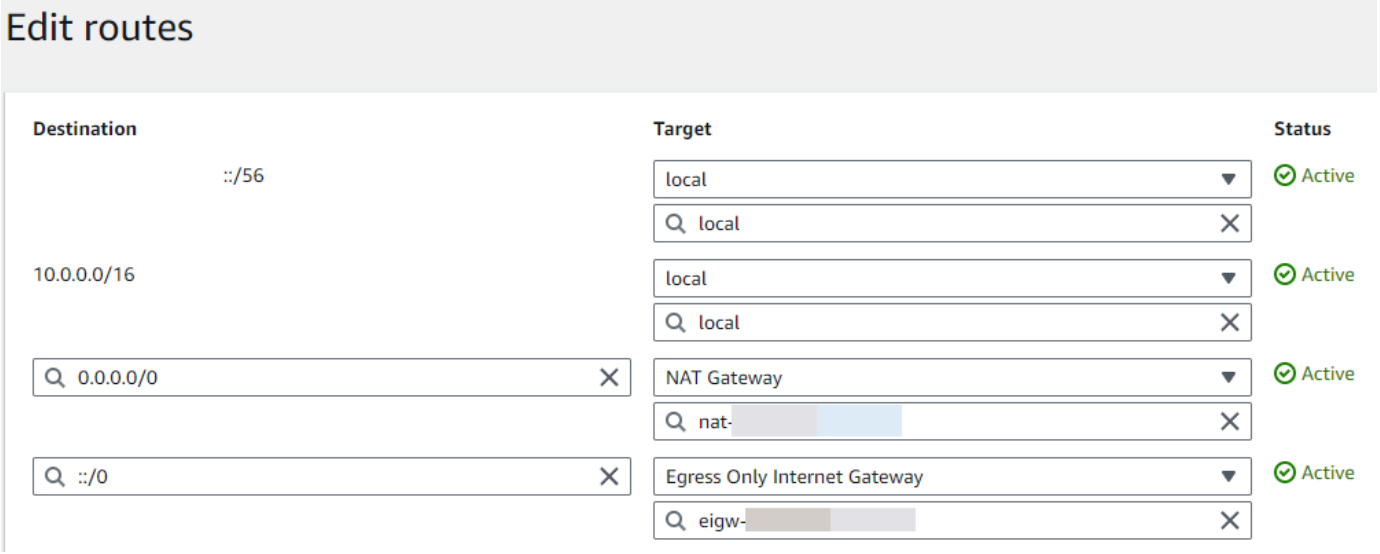
11. Sous Routes, choisissez la Modifier des routes.



12. Choisissez Ajouter une route, puis saisissez `::/0` dans la zone Destination.



13. Pour Cible, sélectionnez Passerelle Internet de sortie uniquement, puis choisissez la passerelle que vous avez créée précédemment.



14. Sélectionnez Enregistrer les modifications.

Configurer la fonction Lambda

Pour configurer un VPC lorsque vous créez une fonction

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez Créer une fonction.
3. Sous Informations de base, dans Nom de fonction, entrez un nom pour votre fonction.
4. Développez Advanced settings (Paramètres avancés).
5. Sélectionnez Activer le VPC, puis choisissez un VPC.
6. (Facultatif) Pour autoriser le trafic IPv6 sortant, sélectionnez Autoriser le trafic IPv6 pour les sous-réseaux à double pile.
7. Pour Sous-réseaux, sélectionnez tous les sous-réseaux privés. Les sous-réseaux privés peuvent accéder à Internet via une passerelle NAT. La connexion d'une fonction à un sous-réseau public ne lui donne pas accès à Internet.

   > **Note**
   >
   > Si vous avez sélectionné Autoriser le trafic IPv6 pour les sous-réseaux à double pile, tous les sous-réseaux sélectionnés doivent avoir un bloc CIDR IPv4 et un bloc CIDR IPv6.

8. Pour les groupes de sécurité, sélectionnez un groupe de sécurité qui autorise le trafic sortant.
9. Choisissez Créer une fonction.

Lambda crée automatiquement un rôle d'exécution avec la politique AWSLambdaVPCAccessExecutionRole gérée par AWS. Les autorisations de cette politique ne sont nécessaires que pour créer des interfaces réseau élastiques pour la configuration VPC, et non pour invoquer votre fonction. Pour appliquer les autorisations de moindre privilège, vous pouvez supprimer la politique AWSLambdaVPCAccessExecutionRole de votre rôle d'exécution après avoir créé la fonction et la configuration VPC. Pour de plus amples informations, consultez Autorisations IAM requises.

Pour configurer un VPC pour une fonction existante

Pour ajouter une configuration VPC à une fonction existante, le rôle d'exécution de la fonction doit être autorisé à créer et à gérer des interfaces réseau élastiques. Ces autorisations sont incluses dans la politique AWSLambdaVPCAccessExecutionRole gérée AWS. Pour appliquer les autorisations de

moindre privilège, vous pouvez supprimer la politique AWSLambdaVPCAccessExecutionRole de votre rôle d'exécution après avoir créé la configuration VPC.

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Choisissez l'onglet Configuration puis choisissez VPC.
4. Sous VPC, choisissez Modifier.
5. Sélectionnez le VPC.
6. (Facultatif) Pour autoriser le trafic IPv6 sortant, sélectionnez Autoriser le trafic IPv6 pour les sous-réseaux à double pile.
7. Pour Sous-réseaux, sélectionnez tous les sous-réseaux privés. Les sous-réseaux privés peuvent accéder à Internet via une passerelle NAT. La connexion d'une fonction à un sous-réseau public ne lui donne pas accès à Internet.

   > **Note**
   >
   > Si vous avez sélectionné Autoriser le trafic IPv6 pour les sous-réseaux à double pile, tous les sous-réseaux sélectionnés doivent avoir un bloc CIDR IPv4 et un bloc CIDR IPv6.

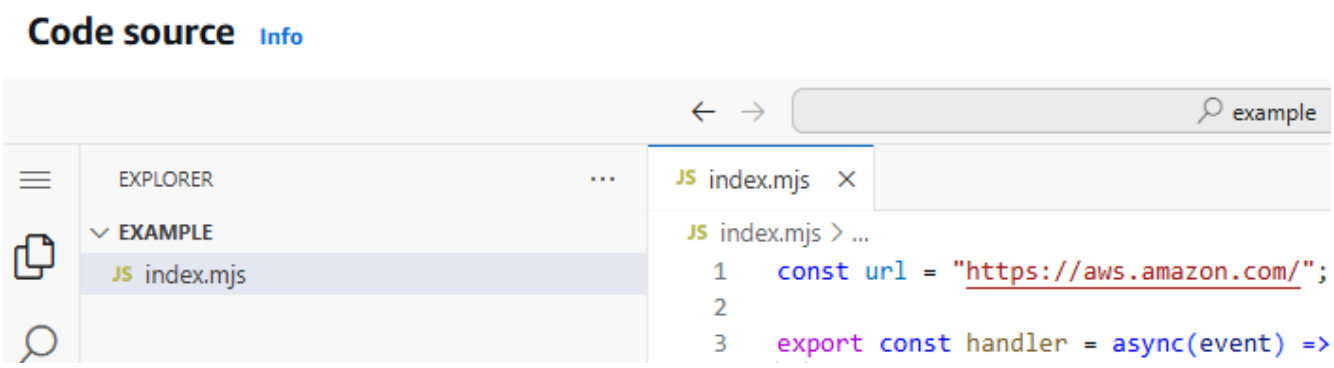
8. Pour les groupes de sécurité, sélectionnez un groupe de sécurité qui autorise le trafic sortant.
9. Choisissez Enregistrer.

Tester la fonction

Utilisez l'exemple de code suivant pour vérifier que votre fonction connectée au VPC peut accéder à l'Internet public. En cas de succès, le code renvoie un code d'état `200`. En cas d'échec, la fonction expire.

Node.js

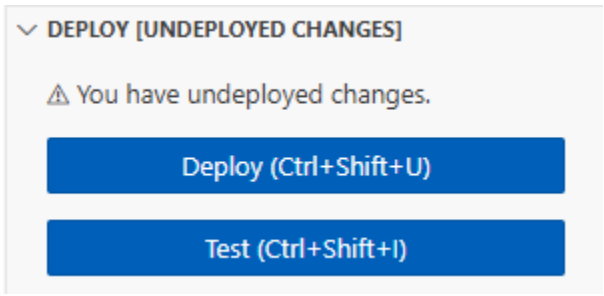
1. Dans le volet Source du code de la console Lambda, collez le code suivant dans le fichier index.mjs. La fonction envoie une requête HTTP GET à un point de terminaison public et renvoie le code de réponse HTTP pour tester si la fonction a accès à l'Internet public.



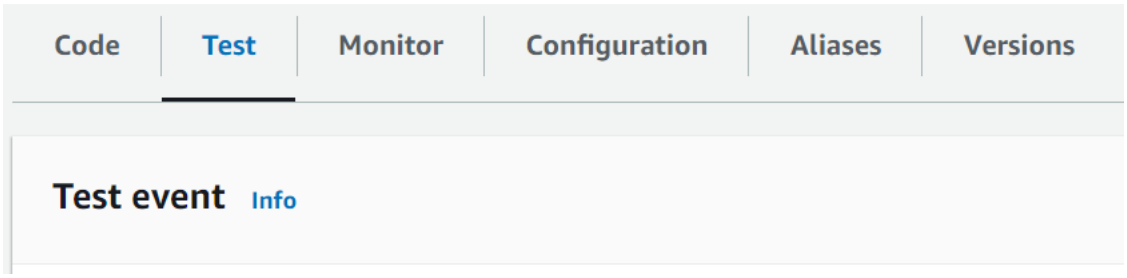
Example – Requête HTTP avec async/await

```
const url = "https://aws.amazon.com/";

export const handler = async(event) => {
    try {
        const res = await fetch(url);
        console.info("status", res.status);
        return res.status;
    }
    catch (e) {
        console.error(e);
        return 500;
    }
};
```

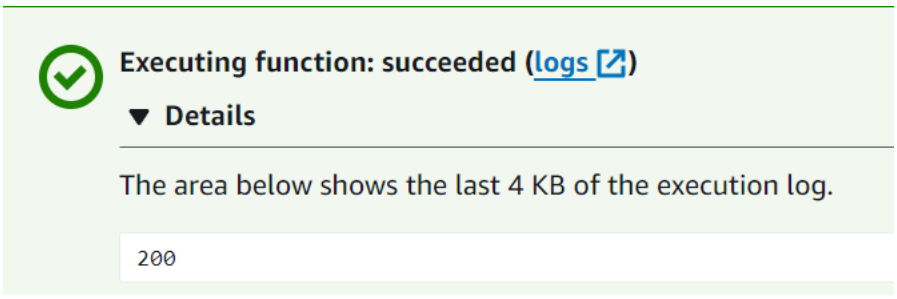
2. Dans la section DÉPLOYER, choisissez Déployer pour mettre à jour le code de votre fonction :



3. Choisissez l'onglet Test.



4. Sélectionnez Tester).
5. La fonction renvoie un code d'état `200`. Cela signifie que la fonction dispose d'un accès Internet sortant.

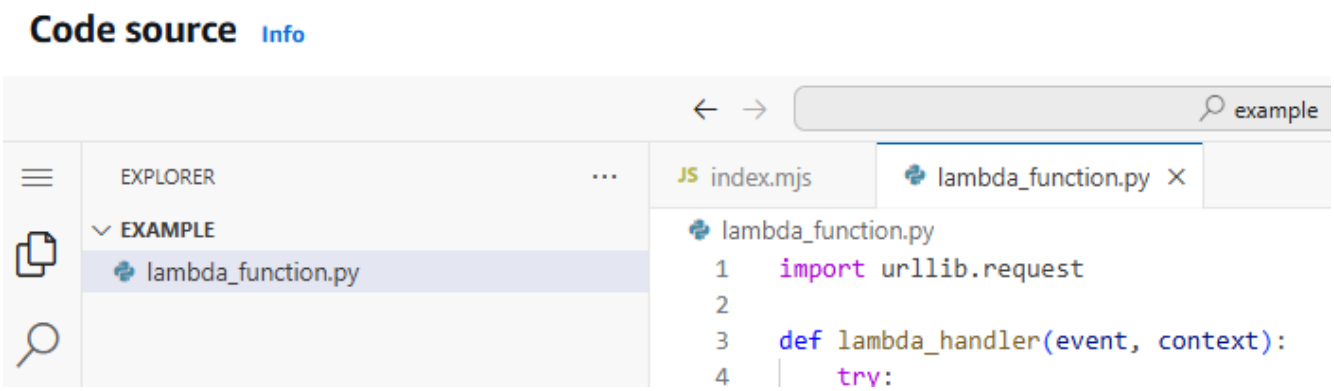


Si la fonction ne parvient pas à accéder à l'Internet public, un message d'erreur de ce type s'affiche :

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

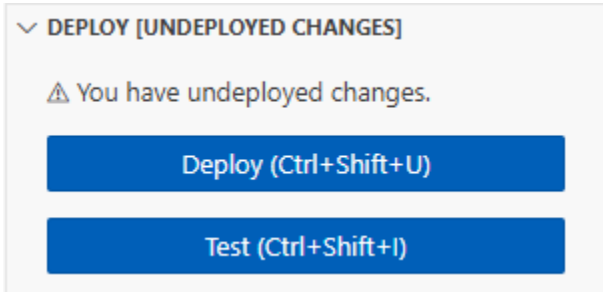
Python

1. Dans le volet Source du code de la console Lambda, collez le code suivant dans le fichier lambda_function.py. La fonction envoie une requête HTTP GET à un point de terminaison public et renvoie le code de réponse HTTP pour tester si la fonction a accès à l'Internet public.

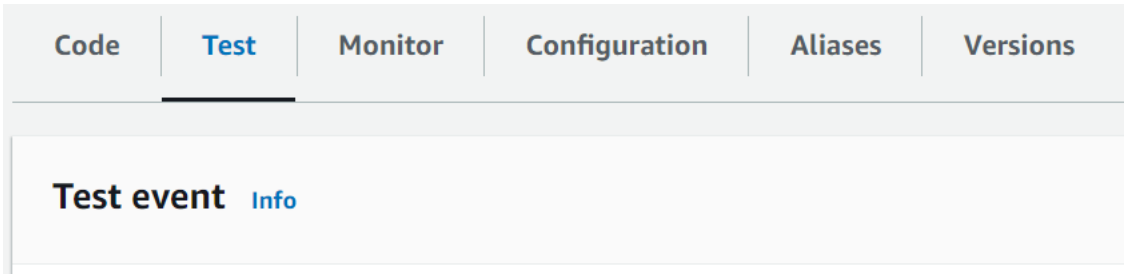



```
import urllib.request

def lambda_handler(event, context):
    try:
        response = urllib.request.urlopen('https://aws.amazon.com')
        status_code = response.getcode()
        print('Response Code:', status_code)
        return status_code
    except Exception as e:
        print('Error:', e)
        raise e
```

2. Dans la section DÉPLOYER, choisissez Déployer pour mettre à jour le code de votre fonction :

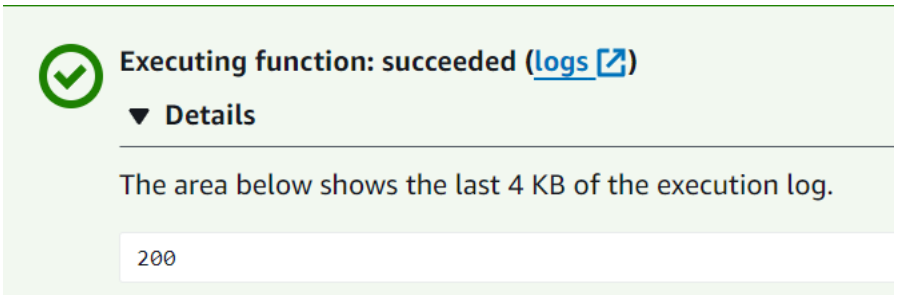


3. Choisissez l'onglet Test.



4. Sélectionnez Tester).

5. La fonction renvoie un code d'état `200`. Cela signifie que la fonction dispose d'un accès Internet sortant.



Si la fonction ne parvient pas à accéder à l'Internet public, un message d'erreur de ce type s'affiche :

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

Connexion des points de terminaison d'un VPC de l'interface entrante pour Lambda

Si vous utilisez Amazon Virtual Private Cloud (Amazon VPC) pour héberger vos ressources AWS, vous pouvez établir une connexion entre votre VPC et Lambda. Vous pouvez utiliser cette connexion pour invoquer votre fonction Lambda sans traverser l'Internet public.

Pour établir une connexion privée entre votre VPC et Lambda, créez un point de terminaison de VPC d'interface. Les points de terminaison d'interface étant alimentés par AWS PrivateLink, vous pouvez accéder en privé aux API Lambda sans passerelle Internet, périphérique NAT, connexion VPN ou connexion AWS Direct Connect. Les instances dans votre VPC n'ont pas besoin d'adresses IP publiques pour communiquer avec des API Lambda. Le trafic entre votre VPC et Lambda ne quitte pas le réseau AWS.

Chaque point de terminaison d'interface est représenté par une ou plusieurs interfaces réseau Elastic dans vos sous-réseaux. Une interface réseau fournit une adresse IP privée qui sert de point d'entrée au trafic vers Lambda.

Sections

- Considérations relatives aux points de terminaison d'interface Lambda
- Création d'un point de terminaison d'interface pour Lambda
- Création d'une stratégie de point de terminaison d'interface pour Lambda

Considérations relatives aux points de terminaison d'interface Lambda

Avant de configurer un point de terminaison d'interface pour Lambda, consulter Propriétés et limites des points de terminaison d'interface dans le Guide de l'utilisateur Amazon VPC.

Vous pouvez appeler n'importe quelle opération d'API Lambda à partir de votre VPC. Par exemple, vous pouvez invoquer la fonction Lambda en appelant l'API `Invoke` à partir de votre VPC. Pour obtenir la liste complète des API Lambda, consultez Actions dans la Référence API Lambda.

`use1-az3` est une région à capacité limitée pour les fonctions Lambda VPC. Vous ne devez pas utiliser de sous-réseaux dans cette zone de disponibilité avec vos fonctions Lambda, car cela peut réduire la redondance zonale en cas de panne.

Keep-alive pour les connexions persistantes

Lambda purgeant les connexions inactives au fil du temps, vous devez utiliser une directive keep-alive pour maintenir les connexions persistantes. Si vous tentez de réutiliser une connexion inactive lorsque vous invoquez une fonction, cela entraîne une erreur de connexion. Pour maintenir votre connexion persistante, utilisez la directive Keep-alive associée à votre environnement d'exécution. Pour obtenir un exemple, voir Réutilisation des connexions avec Keep-Alive dans Node.js dans le Guide du développeur AWS SDK pour JavaScript.

Considérations sur la facturation

L'accès à une fonction Lambda via un point de terminaison d'interface n'occasionne aucun coût supplémentaire. Pour plus d'informations sur la tarification de Lambda, consultez Tarification AWS Lambda.

La tarification standard pour AWS PrivateLink s'applique aux points de terminaison d'interface pour Lambda. Votre compte AWS est facturé pour chaque heure d'approvisionnement de point de terminaison d'interface dans chaque zone de disponibilité, et pour les données traitées via le point de terminaison d'interface. Pour plus d'informations sur la tarification des points de terminaison d'interface, consultez Tarification AWS PrivateLink.

Considérations sur l'appairage de VPC

Vous pouvez connecter d'autres VPC au VPC avec des points de terminaison d'interface à l'aide d'un Appairage de VPC. L'appairage de VPC est une connexion réseau entre deux VPC. Vous pouvez aussi établir une connexion d'appairage de VPC entre vos deux VPC, ou avec un VPC situé dans un autre compte AWS. Les VPC peuvent également se trouver dans deux régions AWS différentes.

Le trafic entre des VPC appairés reste sur le réseau AWS et ne traverse pas l'Internet public. Une fois les VPC appairés, des ressources telles que des instances Amazon Elastic Compute Cloud (Amazon EC2), des instances Amazon Relational Database Service (Amazon RDS) ou des fonctions Lambda API basées sur un VPC dans les deux VPC peuvent accéder à l'API Lambda via des points de terminaison d'interface créés dans l'un des VPC.

Création d'un point de terminaison d'interface pour Lambda

Vous pouvez créer un point de terminaison d'interface pour Lambda à l'aide de la console Amazon VPC ou de l'AWS Command Line Interface (AWS CLI). Pour de plus amples informations, veuillez consulter Création d'un point de terminaison d'interface dans le Amazon VPC Guide de l'utilisateur.

Pour créer un point de terminaison d'interface pour Lambda (console)

1. Ouvrez la page Points de terminaison de la console Amazon VPC.
2. Choisissez Créer un point de terminaison.
3. Pour Catégorie de service, assurez-vous que Services AWS est sélectionné.
4. Pour Nom du service, choisissez com.amazonaws.***région***.lambda. Vérifiez que le Type est Interface.
5. Choisissez un VPC et des sous-réseaux
6. Pour activer un DNS privé pour le point de terminaison d'interface, sélectionnez la case à cocher Enable DNS Name (Activer le nom de DNS). Nous vous recommandons d'activer des noms d'hôtes DNS privés pour votre point de terminaison de VPC pour les Services AWS. Cela garantit que les requêtes qui utilisent les points de terminaison du service public, telles que les requêtes effectuées via un kit SDK AWS, sont résolues vers votre point de terminaison de VPC.
7. Pour Groupes de sécurité, choisissez un ou plusieurs groupes de sécurité.
8. Choisissez Créer un point de terminaison.

Pour utiliser l'option de DNS privée, vous devez définir les `enableDnsHostnames` et `enableDnsSupportattributes` de votre VPC. Pour plus d'informations, consultez Affichage et mise à jour de la prise en charge de DNS pour votre VPC dans le Guide de l'utilisateur Amazon VPC. Si vous activez le DNS privé pour le point de terminaison d'interface, vous pouvez adresser des demandes d'API à Lambda en utilisant son nom DNS par défaut pour la région, par exemple, `lambda.us-east-1.amazonaws.com`. Pour plus de points de terminaison de service, consultez Points de terminaison de service et quotas dans le Références générales AWS.

Pour de plus amples informations, veuillez consulter Accès à un service via un point de terminaison d'interface dans le Amazon VPC Guide de l'utilisateur.

Pour plus d'informations sur la création et la configuration d'un point de terminaison avec CloudFormation, consultez la ressource AWS::EC2::VPCEndpoint dans le Guide de l'utilisateur AWS CloudFormation.

Pour créer un point de terminaison d'interface pour Lambda (AWS CLI)

Utilisez la commande create-vpc-endpoint et spécifiez l'ID du VPC, le type du point de terminaison de VPC (interface), le nom du service, les sous-réseaux qui utiliseront le point de terminaison et les groupes de sécurité à associer aux interfaces réseau du point de terminaison. Par exemple :

```
aws ec2 create-vpc-endpoint
  --vpc-id vpc-ec43eb89
  --vpc-endpoint-type Interface
  --service-name com.amazonaws.us-east-1.lambda
  --subnet-id subnet-abababab
  --security-group-id sg-1a2b3c4d
```

Création d'une stratégie de point de terminaison d'interface pour Lambda

Pour contrôler qui peut utiliser votre point de terminaison d'interface, ainsi que les fonctions Lambda auxquelles l'utilisateur peut accéder, vous pouvez attacher une stratégie de point de terminaison à votre point de terminaison. La stratégie spécifie les informations suivantes :

- Le principal qui peut exécuter des actions.
- Les actions que le principal peut effectuer.
- Ressources sur lesquelles le principal peut effectuer des actions.

Pour de plus amples informations, veuillez consulter Contrôle de l'accès aux services avec points de terminaison d'un VPC dans le Amazon VPC Guide de l'utilisateur.

Exemple : stratégie de point de terminaison d'interface pour des actions Lambda

Voici un exemple de stratégie de point de terminaison pour Lambda. Lorsqu'elle est attachée à un point de terminaison, cette stratégie permet à l'utilisateur `MyUser` d'invoquer la fonction `my-function`.

> **Note**
>
> Vous devez inclure les ARN de la fonction qualifiée et de celle non qualifiée dans la ressource.

```
{
   "Statement":[
      {
         "Principal":
         {
             "AWS": "arn:aws:iam::111122223333:user/MyUser"
         },
```

```
         "Effect":"Allow",
         "Action":[
            "lambda:InvokeFunction"
         ],
         "Resource": [
               "arn:aws:lambda:us-east-2:123456789012:function:my-function",
               "arn:aws:lambda:us-east-2:123456789012:function:my-function:*"
            ]
      }
   ]
}
```

Configuration de l'accès au système de fichiers pour les fonctions Lambda

Vous pouvez configurer une fonction pour monter un système de fichiers Amazon Elastic File System (Amazon EFS) dans un répertoire local. Avec Amazon EFS, votre code de fonction peut accéder aux ressources partagées et les modifier en toute sécurité et avec une haute simultanéité.

Régions prises en charge

Amazon EFS pour Lambda est disponible dans toutes les régions commerciales, à l'exception des régions Asie-Pacifique (Nouvelle-Zélande), Asie-Pacifique (Taipei), Asie-Pacifique (Malaisie), Mexique (Centre), Asie-Pacifique (Thaïlande) et Canada Ouest (Calgary).

Sections

- Rôle d'exécution et autorisations utilisateur
- Configuration d'un système de fichiers et d'un point d'accès
- Connexion à un système de fichiers (console)

Rôle d'exécution et autorisations utilisateur

Si le système de fichiers ne possède pas de politique Gestion des identités et des accès AWS (IAM) configurée par l'utilisateur, EFS utilise une politique par défaut qui accorde un accès complet à tout client pouvant se connecter au système de fichiers à l'aide d'une cible de montage du système de fichiers. Si le système de fichiers dispose d'une stratégie IAM configurée par l'utilisateur, le rôle d'exécution de votre fonction doit avoir le bon.`elasticfilesystem`Autorisations.

Autorisations du rôle d'exécution

- elasticfilesystem:ClientMount
- elasticfilesystem:ClientWrite (non requis pour les connexions en lecture seule)

Ces autorisations sont incluses dans la stratégie gérée AmazonElasticFileSystemClientReadWriteAccess . En outre, votre rôle d'exécution doit posséder leautorisations requises pour se connecter au VPC du système de fichiers.

Lorsque vous configurez un système de fichiers, Lambda utilise vos autorisations pour vérifier les cibles de montage. Pour configurer une fonction pour vous connecter à un système de fichiers, votre utilisateur a besoin des autorisations suivantes :

Autorisations des utilisateurs

- elasticfilesystem:DescribeMountTargets

Configuration d'un système de fichiers et d'un point d'accès

Créez un système de fichiers Amazon EFS avec une cible de montage dans chaque zone de disponibilité à laquelle votre fonction se connecte. Pour les performances et une résilience, utilisez au moins deux zones de disponibilité. Par exemple, dans une configuration simple, vous pouvez avoir un VPC avec deux sous-réseaux privés dans des zones de disponibilité distinctes. La fonction se connecte aux deux sous-réseaux et une cible de montage est disponible dans chacun. Assurez-vous que le trafic NFS (port 2049) est autorisé par les groupes de sécurité utilisés par la fonction et les cibles de montage.

> **Note**
>
> Lorsque vous créez un système de fichiers, vous choisissez un mode de performances qui ne peut pas être modifié ultérieurement. Le mode Usage général a une latence plus faible, et le mode I/O max prend en charge un débit maximal et des I/O par seconde plus élevées. Pour orienter votre choix, consultez Performances d'Amazon EFS dans le Guide de l'utilisateur Amazon Elastic File System.

Un point d'accès relie chaque instance de la fonction à la cible de montage droite pour la zone de disponibilité à laquelle elle se connecte. Pour des performances optimales, créez un point d'accès avec un chemin non racine et limitez le nombre de fichiers que vous créez dans chaque répertoire. L'exemple suivant crée un répertoire nommé `my-function` sur le système de fichiers et définit l'ID propriétaire sur 1001 avec les autorisations d'annuaire standard (755).

Example configuration du point d'accès

- Nom – `files`
- ID de l'utilisateur – `1001`
- ID du groupe – `1001`

- Chemin – `/my-function`
- Permissions (Autorisations – `755`
- ID d'utilisateur du propriétaire – `1001`
- ID d'utilisateur du groupe – `1001`

Lorsqu'une fonction utilise le point d'accès, elle reçoit l'ID utilisateur 1001 et dispose d'un accès complet au répertoire.

Pour plus d'informations, consultez les rubriques suivantes dans le Guide de l'utilisateur Amazon Elastic File System User.

- Création de ressources pour Amazon EFS
- Collaboration avec des utilisateurs, des groupes et des autorisations

Connexion à un système de fichiers (console)

Une fonction se connecte à un système de fichiers sur le réseau local dans un VPC. Les sous-réseaux auquel votre fonction se connecte peuvent être les mêmes sous-réseaux qui contiennent des points de montage pour votre système de fichiers, ou des sous-réseaux de la même zone de disponibilité qui peuvent acheminer le trafic NFS (port 2049) vers le système de fichiers.

> **Note**
>
> Si votre fonction n'est pas déjà connectée à un VPC, veuillez consulter Octroi aux fonctions Lambda d'un accès aux ressources d'un Amazon VPC.

Pour configurer l'accès au système de fichiers

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Sélectionnez Configuration, puis File systems (Systèmes de fichiers).
4. Sous Système de fichiers, choisissez Ajouter un système de fichiers.
5. Configurez les propriétés suivantes :
   - Système de fichiers EFS – Point d'accès d'un système de fichiers dans le même VPC.

- Chemin de montage local – Emplacement où le système de fichiers est monté sur la fonction Lambda, commençant par `/mnt/`.

> **Tarification**
>
> Amazon EFS facture le stockage et le débit, avec des tarifs qui varient selon la classe de stockage. Pour plus d'informations, consultez Tarification Amazon EFS.
> Lambda facture le transfert de données entre VPC. Cela ne s'applique que si le VPC de votre fonction est appairé à un autre VPC avec un système de fichiers. Les tarifs sont les mêmes que pour le transfert de données Amazon EC2 entre VPC dans une même région. Pour plus d’informations, consultez Tarification ambda.

Création d'un alias de fonction Lambda

Vous pouvez créer des alias pour votre fonction Lambda. Un alias Lambda est un pointeur vers une version de la fonction que vous pouvez mettre à jour. Les utilisateurs de la fonction peuvent accéder à la version de la fonction en utilisant l'alias Amazon Resource Name (ARN). Lorsque vous déployez une nouvelle version, vous pouvez mettre à jour l'alias pour utiliser la nouvelle version, ou diviser le trafic entre deux versions.

Console

Pour créer un alias à l'aide de la console

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Sélectionnez Alias, puis Create alias (Créer un alias).
4. Sur la page Create alias (Créer un alias), procédez de la manière suivante :
   a. Renseignez le champ Name (Nom) pour l'alias.
   b. (Facultatif) Renseignez le champ Description de l'alias.
   c. Pour Version, choisissez une version de fonction vers laquelle vous souhaitez que l'alias pointe.
   d. (Facultatif) Pour configurer le routage sur l'alias, développez Weighted alias (Alias pondéré). Pour de plus amples informations, veuillez consulter Mise en œuvre de déploiements canary Lambda à l'aide d'un alias pondéré.
   e. Choisissez Enregistrer.

AWS CLI

Pour créer un alias à l'aide de l'AWS Command Line Interface (AWS CLI), utilisez la commande create-alias.

```
aws lambda create-alias \
  --function-name my-function \
  --name alias-name \
  --function-version version-number \
  --description " "
```

Pour modifier un alias afin qu'il pointe vers une nouvelle version de la fonction, utilisez la commande update-alias.

```
aws lambda update-alias \
  --function-name my-function \
  --name alias-name \
  --function-version version-number
```

Pour créer un alias, utilisez la commande delete-alias.

```
aws lambda delete-alias \
  --function-name my-function \
  --name alias-name
```

Les commandes AWS CLI dans les étapes précédentes correspondent aux opérations d'API Lambda suivantes :

- CreateAlias
- UpdateAlias
- DeleteAlias

Utilisation d’alias Lambda dans les sources d’événements et les politiques d’autorisation

Chaque alias a un ARN unique. Un alias peut uniquement pointer vers une version de fonction, et non vers un autre alias. Vous pouvez mettre à jour un alias de sorte qu’il pointe vers une nouvelle version de la fonction.

Des sources d'événements telles qu'Amazon Simple Storage Service (Amazon S3) appellent votre fonction Lambda. Ces sources d'événements gèrent un mappage qui identifie la fonction à appeler lorsque des événements se produisent. Si vous spécifiez un alias de fonction Lambda dans la configuration du mappage, vous n'avez pas besoin de mettre à jour le mappage lorsque la version de la fonction change. Pour de plus amples informations, veuillez consulter Procédure de traitement par Lambda des enregistrements provenant de sources d’événements basées sur des flux et des files d’attente.

Dans une stratégie de ressources, vous pouvez accorder des autorisations en lien avec des sources d'événements à utiliser votre fonction Lambda. Si vous spécifiez un ARN d'alias dans la stratégie, vous n'avez pas besoin de mettre à jour la stratégie lorsque la version de la fonction change.

Stratégies basées sur une ressource

Vous pouvez utiliser une stratégie basée sur les ressources pour accorder l'accès à votre fonction à un service, à une ressource ou à un compte. La portée de cette autorisation est dépendante du fait que vous appliquiez l'autorisation à un alias, à une version ou à la fonction entière. Par exemple, si vous utilisez un nom d'alias (tel que `helloworld:PROD`), l'autorisation vous permet d'appeler la fonction `helloworld` à l'aide de l'ARN d'alias (`helloworld:PROD`).

Si vous essayez d'appeler la fonction sans alias ou version spécifique, vous obtenez une erreur d'autorisation. Cette erreur d'autorisation continue de se produire même si vous tentez d'appeler directement la version de fonction associée à l'alias.

Par exemple, la commande AWS CLI suivante accorde des autorisations Amazon S3 pour appeler l'alias PROD de la fonction `helloworld` quand Amazon S3 agit pour le compte de `amzn-s3-demo-bucket`.

```
aws lambda add-permission \
  --function-name helloworld \
  --qualifier PROD \
```

```
  --statement-id 1 \
  --principal s3.amazonaws.com \
  --action lambda:InvokeFunction \
  --source-arn arn:aws:s3:::amzn-s3-demo-bucket \
  --source-account 123456789012
```

Pour de plus amples informations sur l'utilisation des noms de ressources dans les stratégies, veuillez consulter Optimisation des sections relatives aux stratégies de Ressources et Conditions.

Mise en œuvre de déploiements canary Lambda à l'aide d'un alias pondéré

Vous pouvez utiliser un alias pondéré pour répartir le trafic entre deux versions différentes de la même fonction. Cette approche vous permet de tester de nouvelles versions de vos fonctions avec un faible pourcentage de trafic et de revenir rapidement en arrière si nécessaire. C'est ce que l'on appelle un déploiement canary. Les déploiements Canary diffèrent blue/green des déploiements en exposant la nouvelle version à une partie seulement des demandes plutôt que de changer tout le trafic en une seule fois.

Vous pouvez faire pointer un alias vers au maximum deux versions de fonction Lambda. Les versions doivent répondre aux critères suivants :

- Les deux versions doivent avoir le même rôle d'exécution.
- Les deux versions doivent avoir la même configuration de file d'attente de lettres mortes ou aucune configuration de file d'attente de lettres mortes.
- Les deux versions doivent être publiées. L'alias ne peut pas pointer vers `$LATEST`.

> **Note**
>
> Lambda utilise un modèle probabiliste simple pour distribuer le trafic entre les deux versions de la fonction. Quand le niveau de trafic est faible, il se peut que vous observiez une variance élevée entre les pourcentages de trafic configuré et réel sur chaque version. Si votre fonction utilise une simultanéité approvisionnée, vous pouvez éviter des invocations de débordement en configurant un plus grand nombre d'instances de simultanéité approvisionnées pendant que le routage d'alias est actif.

Création d'un alias pondéré

Console

Pour configurer le routage sur un alias à l'aide de la console

> **Note**
>
> Vérifiez que la fonction a au moins deux versions publiées. Pour créer des versions supplémentaires, suivez les instructions de la section Création de versions de fonction.

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Sélectionnez Alias, puis Create alias (Créer un alias).
4. Sur la page Create alias (Créer un alias), procédez de la manière suivante :
   a. Renseignez le champ Name (Nom) pour l'alias.
   b. (Facultatif) Renseignez le champ Description de l'alias.
   c. Pour Version, choisissez la première version de fonction vers laquelle vous souhaitez que l'alias pointe.
   d. Développez Weighted alias (Alias pondéré).
   e. Pour Additional version (Version supplémentaire), choisissez la deuxième version de fonction vers laquelle vous souhaitez que l'alias pointe.
   f. Pour Weight (%) (Pondération (%)), entrez une valeur de pondération pour la fonction. La pondération est le pourcentage de trafic qui est affecté à cette version lorsque l'alias est appelé. La première version reçoit le reste du trafic. Par exemple, si vous spécifiez 10 % pour Additional version (Version supplémentaire), 90 % du trafic est automatiquement affecté à la première version.
   g. Choisissez Enregistrer.

AWS CLI

Utilisez les AWS CLI commandes create-alias et update-alias pour configurer les pondérations du trafic entre deux versions de fonction. Lorsque vous créez ou mettez à jour l'alias, vous spécifiez la pondération du trafic dans le paramètre `routing-config`.

L'exemple suivant crée un alias de fonction Lambda nommé routing-alias qui pointe vers la version 1 de la fonction. La version 2 de la fonction reçoit 3 % du trafic. Les 97 % restants du trafic sont acheminés vers la version 1.

```
aws lambda create-alias \
  --name routing-alias \
  --function-name my-function \
  --function-version 1  \
  --routing-config AdditionalVersionWeights={"2"=0.03}
```

Utilisez la commande `update-alias` pour augmenter le pourcentage de trafic entrant vers la version 2. Dans l'exemple suivant, vous augmentez le trafic à 5 %.

```
aws lambda update-alias \
  --name routing-alias \
  --function-name my-function \
  --routing-config AdditionalVersionWeights={"2"=0.05}
```

Pour acheminer l'ensemble du trafic vers la version 2, utilisez la commande `update-alias` pour modifier la propriété `function-version` afin que l'alias pointe vers la version 2. La commande réinitialise également la configuration de routage.

```
aws lambda update-alias \
  --name routing-alias \
  --function-name my-function  \
  --function-version 2 \
  --routing-config AdditionalVersionWeights={}
```

Les AWS CLI commandes des étapes précédentes correspondent aux opérations d'API Lambda suivantes :

- CreateAlias
- UpdateAlias

Détermination de la version invoquée

Lorsque vous configurez des pondérations de trafic entre deux versions de fonction, vous pouvez déterminer la version de fonction Lambda appelée de deux manières :

- CloudWatch Journaux — Lambda émet automatiquement une entrée de `START` journal contenant l'ID de version invoqué pour chaque appel de fonction. Exemple :

  ```
  START RequestId: 1dh194d3759ed-4v8b-a7b4-1e541f60235f Version: 2
  ```

  Pour les appels d'alias, Lambda utilise la dimension `ExecutedVersion` pour filtrer les métriques en fonction de la version appelée. Pour de plus amples informations, veuillez consulter Affichage des métriques pour une fonction Lambda.
- Charge utile de réponse (appels synchrones) – Les réponses aux appels de fonction synchrones incluent un en-tête `x-amz-executed-version` indiquant la version de fonction appelée.

Création d'un déploiement continu avec des alias pondérés

Utilisez AWS CodeDeploy and AWS Serverless Application Model (AWS SAM) pour créer un déploiement continu qui détecte automatiquement les modifications apportées à votre code de fonction, déploie une nouvelle version de votre fonction et augmente progressivement le volume de trafic vers la nouvelle version. La quantité de trafic et le taux d'augmentation sont des paramètres que vous pouvez configurer.

Dans le cadre d'un déploiement continu, AWS SAM exécute les tâches suivantes :

- Il configure votre fonction Lambda et crée un alias. La configuration de routage d'alias pondéré est la capacité sous-jacente qui implémente le déploiement propagé.
- Crée une CodeDeploy application et un groupe de déploiement. Le groupe de déploiement gère le déploiement propagé et la restauration, si nécessaire.
- Il détecte le moment où vous créez une nouvelle version de votre fonction Lambda.
- Déclencheurs CodeDeploy permettant de démarrer le déploiement de la nouvelle version.

Exemple de AWS SAM modèle

L'exemple suivant présente un modèle AWS SAM pour un déploiement propagé simple.

```
AWSTemplateFormatVersion : '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: A sample SAM template for deploying Lambda functions

Resources:
# Details about the myDateTimeFunction Lambda function
```

```
  myDateTimeFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: myDateTimeFunction.handler
      Runtime: nodejs24.x
# Creates an alias named "live" for the function, and automatically publishes when you
 update the function.
      AutoPublishAlias: live
      DeploymentPreference:
# Specifies the deployment configuration
          Type: Linear10PercentEvery2Minutes
```

Ce modèle définit une fonction Lambda nommée `myDateTimeFunction` avec les propriétés suivantes.

AutoPublishAlias

La propriété `AutoPublishAlias` crée un alias nommé `live`. De plus, l'infrastructure AWS SAM détecte automatiquement le moment où vous enregistrez du code nouveau pour la fonction. L'infrastructure publie ensuite une nouvelle version de fonction et met à jour l'alias `live` de sorte qu'il pointe vers la nouvelle version.

DeploymentPreference

La `DeploymentPreference` propriété détermine la vitesse à laquelle l' CodeDeploy application déplace le trafic de la version d'origine de la fonction Lambda vers la nouvelle version. La valeur `Linear10PercentEvery2Minutes` déplace 10 % du trafic toutes les deux minutes vers la nouvelle version.

Pour obtenir la liste des configurations de déploiement prédéfinies, consultez Configurations de déploiement.

Pour plus d'informations sur la façon de créer des déploiements progressifs avec CodeDeploy et AWS SAM, consultez les rubriques suivantes :

- Tutoriel : Déployer une fonction Lambda mise à jour avec et CodeDeploy AWS Serverless Application Model
- Déploiement progressif d'applications sans serveur avec AWS SAM

Gestion des versions d'une fonction Lambda

Vous pouvez utiliser des versions pour gérer le déploiement de vos fonctions. Par exemple, vous pouvez publier une nouvelle version d'une fonction à des fins de test bêta sans affecter les utilisateurs de la version de production stable. Lambda crée une nouvelle version de votre fonction chaque fois que vous la publiez. La nouvelle version est une copie de la version non publiée de la fonction. La version non publiée est nommée `$LATEST`.

Il est important de noter que chaque fois que vous déployez le code de votre fonction, vous remplacez le code actuel dans `$LATEST`. Pour enregistrer l'itération `$LATEST` actuelle, créez une nouvelle version de la fonction. Si `$LATEST` est identique à une version publiée précédemment, aucune nouvelle version ne pourra être créée tant que les modifications n'auront pas été déployées dans `$LATEST`. Ces modifications peuvent inclure la mise à jour du code ou la modification des paramètres de configuration de la fonction.

Une fois que vous avez publié une version de fonction, son code, son environnement d'exécution, son architecture, sa mémoire, ses couches et la plupart des autres paramètres de configuration sont immuables. Cela signifie que vous ne pouvez pas modifier ces paramètres sans publier une nouvelle version à partir de `$LATEST`. Vous pouvez configurer les éléments suivants pour une version de fonction publiée :

- Déclencheurs
- Destinations
- Simultanéité allouée
- invocation asynchrone
- Connexions aux bases de données et proxys

> **Note**
>
> Lorsque vous utilisez les contrôles de gestion de l'environnement d'exécution avec le mode Auto, la version de l'environnement d'exécution utilisée par la version de la fonction est mise à jour automatiquement. Lorsque vous utilisez le mode Function update (Mise à jour de fonction) ou Manual (Manuel), la version de l'environnement d'exécution n'est pas mise à jour. Pour de plus amples informations, veuillez consulter the section called "Mises à jour de version de l'environnement d'exécution".

Sections

- Création de versions de fonction
- Utilisation des versions
- Octroi d'autorisations

Création de versions de fonction

Vous pouvez modifier le code et les paramètres de la fonction uniquement sur la version non publiée d'une fonction. Lorsque vous publiez une version, Lambda verrouille le code et la plupart des paramètres afin de préserver une expérience cohérente pour les utilisateurs de cette version.

Vous pouvez créer une version de fonction à l'aide de la console Lambda.

Pour créer une version de fonction

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction, puis choisissez l'onglet Versions.
3. Sur la page de configuration des versions, sélectionnez Publish new version (Publier une nouvelle version).
4. (Facultatif) Entrez une description de la version.
5. Choisissez Publier.

Vous pouvez également publier une version d'une fonction à l'aide de l'opération PublishVersionAPI.

La AWS CLI commande suivante publie une nouvelle version d'une fonction. La réponse renvoie les informations de configuration relatives à la nouvelle version, y compris le numéro de version et l'ARN de la fonction avec le suffixe de version.

```
aws lambda publish-version --function-name my-function
```

Vous devriez voir la sortie suivante :

```
{
  "FunctionName": "my-function",
  "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function:1",
  "Version": "1",
```

```
  "Role": "arn:aws:iam::123456789012:role/lambda-role",
  "Handler": "function.handler",
  "Runtime": "nodejs24.x",
  ...
}
```

> **Note**
>
> Lambda attribue des numéros de séquence à croissance monotone pour la gestion des versions. Lambda ne réutilise jamais les numéros de version, même après avoir supprimé et recréé une fonction.

Utilisation des versions

Vous pouvez référencer votre fonction Lambda à l'aide d'un ARN qualifié ou d'un ARN non qualifié.

- ARN qualifié – ARN de la fonction avec suffixe de version. L'exemple suivant fait référence à la version 42 de la fonction `helloworld`.

  ```
  arn:aws:lambda:aws-region:acct-id:function:helloworld:42
  ```

- ARN non qualifié – ARN de la fonction sans suffixe de version.

  ```
  arn:aws:lambda:aws-region:acct-id:function:helloworld
  ```

Vous pouvez utiliser un ARN qualifié ou non qualifié dans toutes les opérations d'API pertinentes. Cependant, vous ne pouvez pas utiliser un ARN non qualifié pour créer un alias.

Si vous décidez de ne pas publier de versions de fonction, vous pouvez invoquer la fonction à l'aide de l'ARN qualifié ou non qualifié dans votre mappage source d'événement. Lorsque vous invoquez une fonction à l'aide d'un ARN non qualifié, Lambda invoque implicitement `$LATEST`.

L'ARN qualifié pour chaque version de fonction Lambda est unique. Après avoir publié une version, vous ne pouvez pas modifier l'ARN ou le code de fonction.

Lambda ne publie une nouvelle version de fonction que si le code n'a jamais été publié ou si le code a changé par rapport à la version publiée la plus récente. S'il n'y a pas de changement, la version de fonction publiée la plus récente reste active.

Lorsque vous publiez une version, Lambda crée un instantané immuable du code et de la configuration de votre fonction. Les modifications de configuration ne déclenchent pas toutes la publication d'une nouvelle version. Les modifications de configuration suivantes rendent une fonction éligible à la publication d'une nouvelle version :

- Code de fonction
- Variables d'environnement
- Environnement d'exécution
- Handler (Gestionnaire)
- Layers
- Taille de la mémoire
- Timeout
- Configuration VPC
- Configuration de file d'attente de lettres mortes (DLQ)
- Rôle IAM
- Description
- Architecture (x86_64 ou arm64)
- Taille du stockage éphémère
- Type de package
- Configuration de la journalisation
- Configuration du système de fichiers
- SnapStart
- Configuration du suivi

Les paramètres opérationnels, comme la simultanéité réservée, ne déclenchent pas la publication d'une nouvelle version lorsqu'ils sont modifiés.

Octroi d'autorisations

Vous pouvez utiliser une stratégie basée sur les ressources ou une stratégie basée sur l'identité pour accorder l'accès à votre fonction. La portée de l'autorisation est dépendante du fait que vous appliquiez la stratégie à une fonction ou à une version d'une fonction. Pour de plus amples informations sur les noms de ressource de fonction dans les stratégies, veuillez consulter Optimisation des sections relatives aux stratégies de Ressources et Conditions.

Vous pouvez simplifier la gestion des sources d'événements et des politiques Gestion des identités et des accès AWS (IAM) en utilisant des alias de fonction. Pour de plus amples informations, veuillez consulter Création d’un alias de fonction Lambda.

Utilisation de balises sur les fonctions Lambda

Vous pouvez étiqueter les fonctions pour organiser et gérer vos ressources. Les balises sont des paires clé-valeur de forme libre associées à vos ressources prises en charge par l'ensemble des ressources Services AWS. Pour plus d'informations sur les cas d'utilisation des balises, consultez la section Stratégies de balisage courantes dans le guide des AWS ressources de balisage et de l'éditeur de balises.

Les balises s'appliquent au niveau de la fonction, pas aux versions ni aux alias. Les balises ne font pas partie de la configuration spécifique à la version qui AWS Lambda crée un instantané du moment où vous publiez une version. Vous pouvez utiliser l'API Lambda pour afficher et mettre à jour les balises. Vous pouvez également afficher et mettre à jour les balises tout en gérant une fonction spécifique dans la console Lambda.

Sections

- Autorisations requises pour l'utilisation des balises
- Utilisation des balises avec la console Lambda
- Utilisation de balises avec AWS CLI

Autorisations requises pour l'utilisation des balises

Pour autoriser une identité Gestion des identités et des accès AWS (IAM) – utilisateur, groupe ou rôle – à afficher ou marquer les ressources, accordez-lui les autorisations correspondantes :

- lambda : ListTags —Lorsqu' une ressource possède des balises, accordez cette autorisation à tous ceux qui ont besoin de `ListTags` l'utiliser. Pour les fonctions balisées, cette autorisation est également nécessaire pour `GetFunction`.
- lambda : TagResource —Accordez cette autorisation à toute personne ayant besoin d'appeler `TagResource` ou d'exécuter un tag lors de la création.

Vous pouvez éventuellement envisager d'accorder également l'UntagResourceautorisation lambda : pour autoriser les `UntagResource` appels à la ressource.

Pour de plus amples informations, veuillez consulter Politiques IAM basées sur l'identité pour Lambda.

Utilisation des balises avec la console Lambda

Vous pouvez utiliser la console Lambda pour créer des fonctions qui comportent des balises, pour ajouter des balises aux fonctions existantes et pour filtrer des fonctions selon les balises ajoutées.

Ajout de balises lors de la création d'une fonction

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez Créer une fonction.
3. Choisissez Author from scratch (Créer à partir de zéro) ou Container image (Image de conteneur).
4. Sous Informations de base, configurez votre fonction. Pour de plus amples informations sur la configuration des fonctions, consultez Configuration des fonctions.
5. Développez Advanced settings (Paramètres avancés) et sélectionnez Enable tags (Activer les balises).
6. Pour cela, choisissez Ajouter une balise, puis saisissez une clé et éventuellement une valeur. Répétez cette étape pour ajouter d'autres balises.
7. Choisissez Créer une fonction.

Pour ajouter des balises à une fonction existante

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom d'une fonction.
3. Sélectionnez Configuration, puis Tags (Balises).
4. Sous Balises, choisissez Gérer les balises.
5. Pour cela, choisissez Ajouter une balise, puis saisissez une clé et éventuellement une valeur. Répétez cette étape pour ajouter d'autres balises.
6. Choisissez Enregistrer.

Pour filtrer des fonctions avec des balises

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Cliquez dans la zone de recherche pour afficher la liste des propriétés de fonction et des clés de balise.

3. Choisissez une clé de balise pour afficher la liste des valeurs utilisées dans la AWS région actuelle.
4. Sélectionnez Utiliser : « tag-name » pour afficher toutes les fonctions étiquetées avec cette touche, ou choisissez un Opérateur pour affiner le filtrage en fonction de la valeur.
5. Sélectionnez votre valeur de balise pour appliquer un filtre combinant la clé et la valeur de la balise.

La barre de recherche prend également en charge la recherche de clés de balise. Saisissez `tag` pour afficher uniquement une liste de clés de balise ou entrez le nom d'une clé pour la rechercher dans la liste.

Utilisation de balises avec AWS CLI

Vous pouvez ajouter et supprimer des balises sur les ressources Lambda existantes, fonctions incluses, avec l'API Lambda. Il est également possible d'ajouter des balises lors de la création d'une fonction, vous permettant ainsi de conserver une ressource balisée tout au long de son cycle de vie.

Mise à jour des balises avec la balise Lambda APIs

Vous pouvez ajouter et supprimer des balises pour les ressources Lambda prises en charge par le biais des opérations TagResourceet de l'UntagResourceAPI.

Vous pouvez appeler ces opérations par l'intermédiaire de l' AWS CLI. Pour ajouter des balises à une ressource existante, utilisez la commande `tag-resource`. Cet exemple ajoute deux balises, l'une avec la clé *`Department`* et l'autre avec la clé*`CostCenter`*.

```
aws lambda tag-resource \
--resource arn:aws:lambda:us-east-2:123456789012:resource-type:my-resource \
--tags Department=Marketing,CostCenter=1234ABCD
```

Pour supprimer des balises, utilisez la commande `untag-resource`. Cet exemple supprime la balise contenant la clé*`Department`*.

```
aws lambda untag-resource --resource arn:aws:lambda:us-east-1:123456789012:resource-
type:resource-identifier \
--tag-keys Department
```

Ajout de balises lors de la création d'une fonction

Pour créer une nouvelle fonction Lambda avec des balises, utilisez l'opération CreateFunctionAPI. Spécifiez le paramètre `Tags`. Vous pouvez appeler cette opération à l'aide de la commande `create-function` de la CLI associée à l'option --tags. Avant d'utiliser le paramètre tags avec `CreateFunction`, vérifiez que votre rôle dispose de l'autorisation de baliser les ressources en plus des autorisations habituelles nécessaires à cette opération. Pour plus d'informations sur les autorisations requises pour l'étiquetage, consultez the section called "Autorisations requises pour l'utilisation des balises". Cet exemple ajoute deux balises, l'une avec la clé *`Department`* et l'autre avec la clé*`CostCenter`*.

```
aws lambda create-function --function-name my-function
--handler index.js --runtime nodejs24.x \
--role arn:aws:iam::123456789012:role/lambda-role \
--tags Department=Marketing,CostCenter=1234ABCD
```

Affichage des balises d'une fonction

Pour afficher les balises associées à une ressource Lambda spécifique, utilisez l'opération d'API `ListTags`. Pour de plus amples informations, veuillez consulter ListTags.

Vous pouvez appeler cette opération à l'aide de la `list-tags` AWS CLI commande en fournissant un ARN (Amazon Resource Name).

```
aws lambda list-tags --resource arn:aws:lambda:us-east-1:123456789012:resource-
type:resource-identifier
```

Vous pouvez afficher les balises appliquées à une ressource spécifique à l'aide de l'opération GetFunctiond'API. Aucune fonctionnalité comparable n'est disponible pour les autres types de ressources.

Vous pouvez appeler cette opération à l'aide de la commande `get-function` de la CLI :

```
aws lambda get-function --function-name my-function
```

Filtrage de ressources par balise

Vous pouvez utiliser le fonctionnement de l' AWS Resource Groups Tagging API GetResourcesAPI pour filtrer vos ressources par balises. L'opération `GetResources` reçoit jusqu'à 10 filtres,

chaque filtre contenant une clé de balise et jusqu'à 10 valeurs de balise. Vous fournissez `GetResources`avec un `ResourceType` pour filtrer par certains types de ressources.

Vous pouvez appeler cette opération à l'aide de la `get-resources` AWS CLI commande. Pour des exemples d'utilisation de `get-resources`, consultez get-resources dans la Référence des commandes de l'AWS CLI.

Streaming de réponses pour les fonctions Lambda

Les fonctions Lambda peuvent renvoyer nativement les charges utiles de réponse aux clients via la fonction Lambda ou à l'aide de l'InvokeWithResponseStreamAPI (via le AWS SDK URLs ou des appels d'API directs). Votre fonction Lambda peut également diffuser des charges utiles de réponse via l'intégration du proxy Amazon API Gateway, qui utilise l'InvokeWithResponseStreamAPI pour appeler votre fonction. Le streaming de réponses peut profiter aux applications sensibles à la latence en améliorant les performances de temps au premier octet (TTFB). En effet, vous pouvez renvoyer des réponses partielles au client dès qu'elles sont disponibles. En outre, les fonctions de diffusion des réponses peuvent renvoyer des données utiles allant jusqu'à 200 Mo, contre un maximum de 6 Mo pour les réponses mises en mémoire tampon. La diffusion d'une réponse signifie également que votre fonction n'a pas besoin de stocker l'intégralité de la réponse en mémoire. Pour les réponses très volumineuses, cela peut réduire la quantité de mémoire que vous devez configurer pour votre fonction.

> **Note**
>
> Le streaming des réponses Lambda n'est pas encore entièrement disponible. Régions AWS Consultez la section AWS Capacités par région du Builder Center pour connaître la disponibilité des fonctionnalités par région.

La vitesse à laquelle Lambda diffuse vos réponses dépend de la taille de la réponse. Le débit de diffusion pour les 6 premiers Mo de réponse de votre fonction n'est pas plafonné. Pour les réponses supérieures à 6 Mo, le reste de la réponse est soumis à une limite de bande passante. Pour plus d'informations sur la bande passante de diffusion, consultez Limitation de la bande passante pour la diffusion des réponses.

Les réponses en streaming entraînent des coûts et les réponses diffusées en continu ne sont ni interrompues ni arrêtées lorsque la connexion du client appelant est interrompue. Les clients seront facturés pour la durée complète des fonctions. Les clients doivent donc faire preuve de prudence lorsqu'ils configurent des délais d'expiration de fonction prolongés.

Lambda prend en charge le streaming des réponses sur les exécutions gérées par Node.js. Pour les autres langages, y compris Python, vous pouvez utiliser un environnement d'exécution personnalisé avec une intégration d'API d'exécution personnalisée pour diffuser les réponses ou utiliser l'adaptateur Web Lambda.

> **Note**
>
> Lorsque vous testez votre fonction via la console Lambda, vous verrez toujours les réponses comme étant mises en mémoire tampon.

Rubriques

- Limitation de la bande passante pour la diffusion des réponses
- Compatibilité du VPC avec la diffusion en continu de réponses
- Écriture de fonctions Lambda compatibles avec le streaming de réponses
- Invocation d'une fonction activée pour le streaming de réponses à l'aide de la fonction Lambda URLs
- Tutoriel : création d’une fonction Lambda de streaming de réponses avec une URL de la fonction

Limitation de la bande passante pour la diffusion des réponses

Les 6 premiers Mo des données utiles de la réponse de votre fonction disposent d’une bande passante non plafonnée. Après cette première rafale, Lambda diffuse votre réponse à un débit maximum de 2. MBps Si les réponses de votre fonction ne dépassent jamais 6 Mo, cette limite de bande passante ne s’applique pas.

> **Note**
>
> Les limites de bande passante s'appliquent uniquement à la charge utile de la réponse de votre fonction, et non à l'accès au réseau par votre fonction.

Le débit de bande passante non plafonnée varie en fonction d'un certain nombre de facteurs, notamment la vitesse de traitement de votre fonction. Vous pouvez normalement vous attendre à un débit supérieur à 2 MBps pour les 6 premiers Mo de réponse de votre fonction. Si votre fonction diffuse une réponse vers une destination située à l'extérieur AWS, le débit de diffusion dépend également de la vitesse de la connexion Internet externe.

Compatibilité du VPC avec la diffusion en continu de réponses

Lorsque vous utilisez des fonctions Lambda dans un environnement VPC, la diffusion en continu des réponses doit tenir compte de certaines considérations importantes :

- La fonction Lambda ne prend URLs pas en charge le streaming des réponses dans un environnement VPC.
- Vous pouvez utiliser le streaming de réponses au sein d'un VPC en appelant votre fonction Lambda via le SDK à l'aide de l' AWS API. `InvokeWithResponseStream` Pour cela, vous devez configurer les points de terminaison VPC appropriés pour Lambda.
- Pour les environnements VPC, vous devez créer un point de terminaison de VPC d'interface pour Lambda afin de permettre la communication entre vos ressources dans le VPC et dans le service Lambda.

Une architecture typique pour la diffusion en continu des réponses dans un VPC peut inclure :

```
Client in VPC -> Interface VPC endpoint for Lambda -> Lambda function -> Response
 streaming back through the same path
```

Écriture de fonctions Lambda compatibles avec le streaming de réponses

L'écriture du gestionnaire pour les fonctions de streaming de réponses est différente des modèles de gestionnaire typiques. Lorsque vous écrivez des fonctions de streaming, assurez-vous de faire ce qui suit :

- Encapsulez votre événement avec le décorateur `awslambda.streamifyResponse()`. L'objet global `awslambda` est fourni par l'environnement d'exécution Node.js de Lambda.
- Terminez la diffusion de manière élégante afin de vous assurer que le traitement des données est terminé.

Configuration d'une fonction gestionnaire pour diffuser les réponses

Pour indiquer à l'exécution que Lambda doit diffuser les réponses de votre fonction, vous devez envelopper votre fonction avec le décorateur `streamifyResponse()`. Cela indique à l'exécution d'utiliser le chemin logique approprié pour le streaming des réponses et permet à la fonction de diffuser les réponses.

Le décorateur `streamifyResponse()` accepte une fonction qui accepte les paramètres suivants :

- `event` : fournit des informations sur l'événement d'invocation de l'URL de la fonction, telles que la méthode HTTP, les paramètres de la requête et le corps de la requête.

- `responseStream` : fournit un flux inscriptible.
- `context` : fournit des méthodes et des propriétés avec des informations sur l'invocation, la fonction et l'environnement d'exécution.

L'objet `responseStream` est un `writableStream` Node.js. Comme pour tout flux de ce type, vous devez utiliser la méthode `pipeline()`.

> **Note**
>
> L'objet global `awslambda` est automatiquement fourni par l'environnement d'exécution Node.js de Lambda et aucune importation n'est requise.

Example gestionnaire compatible avec le streaming de réponses

```
import { pipeline } from 'node:stream/promises';
import { Readable } from 'node:stream';

export const echo = awslambda.streamifyResponse(async (event, responseStream, _context)
 => {
  // As an example, convert event to a readable stream.
  const requestStream = Readable.from(Buffer.from(JSON.stringify(event)));

  await pipeline(requestStream, responseStream);
});
```

Bien que `responseStream` propose la méthode `write()` pour écrire dans le flux, nous vous recommandons d'utiliser `pipeline()` dans la mesure du possible. L'utilisation de `pipeline()` permet de s'assurer que le flux accessible en écriture n'est pas submergé par un flux accessible en lecture plus rapide.

Fin du flux

Assurez-vous de terminer correctement le flux avant le retour du gestionnaire. La méthode `pipeline()` s'en charge automatiquement.

Pour les autres cas d'utilisation, appelez la méthode `responseStream.end()` pour terminer correctement un flux. Cette méthode signale que plus aucune donnée ne doit être écrite dans le flux. Cette méthode n'est pas nécessaire si vous écrivez dans le flux avec `pipeline()` ou `pipe()`.

À partir de Node.js 24, Lambda n'attend plus que les promesses non résolues se terminent après le retour de votre gestionnaire ou la fin du flux de réponses. Si votre fonction dépend d'opérations asynchrones supplémentaires, telles que des temporisations ou des extractions, vous devez `await` les inclure dans votre gestionnaire.

Example Exemple de fin d'un flux avec pipeline()

```
import { pipeline } from 'node:stream/promises';

export const handler = awslambda.streamifyResponse(async (event, responseStream,
 _context) => {
  await pipeline(requestStream, responseStream);
});
```

Example Exemple de fin d'un flux sans pipeline()

```
export const handler = awslambda.streamifyResponse(async (event, responseStream,
 _context) => {
  responseStream.write("Hello ");
  responseStream.write("world ");
  responseStream.write("from ");
  responseStream.write("Lambda!");
  responseStream.end();
});
```

Invocation d'une fonction activée pour le streaming de réponses à l'aide de la fonction Lambda URLs

> **Note**
>
> Votre fonction Lambda peut désormais diffuser les charges utiles des réponses via l'intégration du proxy Amazon API Gateway.

Vous pouvez invoquer des fonctions compatibles avec le streaming de réponses en modifiant le mode d'invocation de l'URL de votre fonction. Le mode d'invocation détermine l'opération d'API que Lambda utilise pour invoquer votre fonction. Les modes d'invocation disponibles sont les suivants :

- `BUFFERED` : il s'agit de l'option par défaut. Lambda invoque votre fonction en utilisant l'opération d'API `Invoke`. Les résultats de l'invocation sont disponibles lorsque la charge utile est complète. La taille de la charge utile maximale est de 6 Mo.
- `RESPONSE_STREAM` : permet à votre fonction de diffuser les résultats de la charge utile au fur et à mesure qu'ils sont disponibles. Lambda invoque votre fonction en utilisant l'opération d'API `InvokeWithResponseStream`. La taille maximale de la charge utile de réponse est de 200 Mo.

Vous pouvez toujours invoquer votre fonction sans streaming de réponses en appelant directement l'opération d'API `Invoke`. Cependant, Lambda diffuse toutes les charges utiles de réponse pour les invocations qui passent par l'URL de la fonction jusqu'à ce que vous changiez le mode d'invocation en `BUFFERED`.

Console

Pour définir le mode d'invocation d'une URL de la fonction (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez le nom de la fonction pour laquelle vous voulez définir le mode d'invocation.
3. Choisissez l'onglet Configuration, puis Function URL (URL de fonction).
4. Sélectionnez Modifier, puis Paramètres supplémentaires.
5. Sous Mode d'invocation, sélectionnez le mode d'invocation de votre choix.
6. Choisissez Enregistrer.

AWS CLI

Pour définir le mode d'invocation d'une URL de fonction (AWS CLI)

```
aws lambda update-function-url-config \
  --function-name my-function \
  --invoke-mode RESPONSE_STREAM
```

CloudFormation

Pour définir le mode d'invocation d'une URL de fonction (CloudFormation)

```
MyFunctionUrl:
  Type: AWS::Lambda::Url
```

```
  Properties:
    AuthType: AWS_IAM
    InvokeMode: RESPONSE_STREAM
```

Pour plus d'informations sur la configuration de la fonction URLs, consultez la section Fonction Lambda. URLs

Tutoriel : création d'une fonction Lambda de streaming de réponses avec une URL de la fonction

Dans ce didacticiel, vous créez une fonction Lambda définie comme archive de fichier ZIP avec un point de terminaison d'URL de la fonction qui renvoie un flux de réponses. Pour plus d'informations sur la configuration de la fonction URLs, consultezFonction URLs.

Conditions préalables

Ce didacticiel suppose que vous avez quelques connaissances des opérations Lambda de base et de la console Lambda. Si ce n'est déjà fait, suivez les instructions fournies dans Créer une fonction Lambda à l'aide de la console pour créer votre première fonction Lambda.

Pour effectuer les étapes suivantes, vous avez besoin de l'AWS CLI version 2. Les commandes et la sortie attendue sont répertoriées dans des blocs distincts :

```
aws --version
```

Vous devriez voir la sortie suivante:

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

Pour les commandes longues, un caractère d'échappement (\) est utilisé pour les fractionner en plusieurs lignes.

Sur Linux et macOS, utilisez votre gestionnaire de shell et de package préféré.

> **Note**
>
> Sous Windows, certaines commandes CLI Bash que vous utilisez couramment avec Lambda (par exemple `zip`) ne sont pas prises en charge par les terminaux intégrés du système

d'exploitation. Installez le sous-système Windows pour Linux afin d'obtenir une version intégrée à Windows d'Ubuntu et Bash. Les exemples de commandes CLI de ce guide utilisent le formatage Linux. Les commandes qui incluent des documents JSON en ligne doivent être reformatées si vous utilisez la CLI Windows.

Créer un rôle d'exécution

Créez le rôle d'exécution qui donne à votre fonction Lambda l'autorisation d'accéder aux ressources AWS .

Pour créer un rôle d'exécution

1. Accédez à la page Roles (Rôles) de la Gestion des identités et des accès AWS console (IAM).
2. Sélectionnez Créer un rôle.
3. Créez un rôle avec les propriétés suivantes :
   - Type d'entité sécurisée – Service AWS
   - Cas d'utilisation – Lambda
   - Permissions (Autorisations – AWSLambdaBasicExecutionRole
   - Nom de rôle – **`response-streaming-role`**

La AWSLambdaBasicExecutionRolepolitique dispose des autorisations dont la fonction a besoin pour écrire des CloudWatch journaux sur Amazon Logs. Après avoir créé le rôle, notez son Amazon Resource Name (ARN). Vous en aurez besoin à l'étape suivante.

Création d'une fonction de streaming de réponses (AWS CLI)

Créez une fonction Lambda de streaming de réponses avec un point de terminaison d'URL de la fonction à l'aide de l' AWS Command Line Interface (AWS CLI).

Pour créer une fonction capable de diffuser des réponses

1. Copiez l'exemple de code suivant dans un fichier nommé `index.js`. Cette fonction diffuse trois réponses, séparées de 1 seconde.

```
exports.handler = awslambda.streamifyResponse(
 async (event, responseStream, _context) => {
```

```
  // Metadata is a JSON serializable JS object. Its shape is not defined here.
  const metadata = {
  statusCode: 200,
  headers: {
   "Content-Type": "application/json",
   "CustomHeader": "outerspace"
  }
  };

  // Assign to the responseStream parameter to prevent accidental reuse of the non-
wrapped stream.
  responseStream = awslambda.HttpResponseStream.from(responseStream, metadata);

  responseStream.write("Streaming with Helper \n");
  await new Promise(r => setTimeout(r, 1000));
  responseStream.write("Hello 0 \n");
  await new Promise(r => setTimeout(r, 1000));
  responseStream.write("Hello 1 \n");
  await new Promise(r => setTimeout(r, 1000));
  responseStream.write("Hello 2 \n");
  await new Promise(r => setTimeout(r, 1000));
  responseStream.end();
  await responseStream.finished();
 }
  );
```

2. Créez un package de déploiement.

```
zip function.zip index.js
```

3. Créez une fonction Lambda à l'aide de la commande `create-function`. Remplacez la valeur de `--role` par l'ARN de rôle de l'étape précédente. Cette commande définit le délai d'expiration de la fonction à 10 secondes, ce qui permet à la fonction de diffuser trois réponses.

```
aws lambda create-function \
  --function-name my-streaming-function \
  --runtime nodejs24.x \
  --zip-file fileb://function.zip \
  --handler index.handler \
  --timeout 10 \
  --role arn:aws:iam::123456789012:role/response-streaming-role
```

Pour créer une URL de la fonction

1. Ajoutez une stratégie basée sur les ressources à votre fonction en accordant les autorisations `lambda:InvokeFunctionUrl` et `lambda:InvokeFunction`. Chaque instruction doit être ajoutée dans une commande distincte. Remplacez la valeur de `--principal` par votre Compte AWS identifiant.

```
aws lambda add-permission \
  --function-name my-streaming-function \
  --action lambda:InvokeFunctionUrl \
  --statement-id UrlPolicyInvokeURL \
  --principal 123456789012 \
  --function-url-auth-type AWS_IAM
```

```
aws lambda add-permission \
    --function-name my-streaming-function \
    --action lambda:InvokeFunction \
    --statement-id UrlPolicyInvokeFunction \
    --principal 123456789012
```

2. Créez un point de terminaison d'URL pour la fonction avec la commande `create-function-url-config`.

```
aws lambda create-function-url-config \
  --function-name my-streaming-function \
  --auth-type AWS_IAM \
  --invoke-mode RESPONSE_STREAM
```

> **Note**
>
> Si un message d'erreur concernant `--invoke-mode` s'affiche, il se peut que vous deviez effectuer une mise à niveau vers une version plus récente d' AWS CLI.

Test du point de terminaison d'URL de fonction

Testez votre intégration en invoquant votre fonction. Vous pouvez ouvrir l'URL de votre fonction dans un navigateur ou utiliser curl.

```
curl --request GET "https://abcdefghijklm7nop7qrs740abcd.lambda-url.us-east-1.on.aws/"
 --user "AKIAIOSFODNN7EXAMPLE" --aws-sigv4 "aws:amz:us-east-1:lambda" --no-buffer
```

Notre URL de la fonction utilise le type d'authentification `IAM_AUTH`. Cela signifie que vous devez signer les demandes à la fois avec votre clé d'accès AWS et votre clé secrète. Dans la commande précédente, remplacez par `AKIAIOSFODNN7EXAMPLE` l'ID de la clé d' AWS accès. Entrez votre clé AWS secrète lorsque vous y êtes invité. Si vous n'avez pas votre clé AWS secrète, vous pouvez utiliser des AWS informations d'identification temporaires à la place.

Vous devriez obtenir une réponse comme celle-ci :

```
Streaming with Helper
Hello 0
Hello 1
Hello 2
```

Nettoyage de vos ressources

Vous pouvez maintenant supprimer les ressources que vous avez créées pour ce didacticiel, sauf si vous souhaitez les conserver. En supprimant AWS les ressources que vous n'utilisez plus, vous évitez des frais inutiles pour votre Compte AWS.

Pour supprimer le rôle d'exécution

1. Ouvrez la page Roles (Rôles) de la console IAM.
2. Sélectionnez le rôle d'exécution que vous avez créé.
3. Sélectionnez Delete (Supprimer).
4. Saisissez le nom du rôle dans le champ de saisie de texte et choisissez Delete (Supprimer).

Pour supprimer la fonction Lambda

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction que vous avez créée.
3. Sélectionnez Actions, Supprimer.
4. Saisissez **`confirm`** dans la zone de saisie de texte et choisissez Delete (Supprimer).

Présentation des méthodes d'invocation des fonctions Lambda

Après avoir déployé votre fonction Lambda, vous pouvez l'invoquer de plusieurs manières :

- La console Lambda : utilisez la console Lambda pour créer rapidement un événement de test pour invoquer votre fonction.
- Le AWS SDK — Utilisez le AWS SDK pour appeler votre fonction par programmation.
- L'API Invoke : utilisez l'API Lambda Invoke pour invoquer directement votre fonction.
- The AWS Command Line Interface (AWS CLI) — Utilisez la `aws lambda invoke` AWS CLI commande pour appeler directement votre fonction depuis la ligne de commande.
- Point de terminaison HTTP (S) d'une URL de fonction : utilisez la fonction URLs pour créer un point de terminaison HTTP (S) dédié que vous pouvez utiliser pour appeler votre fonction.

Toutes ces méthodes sont des méthodes directes pour invoquer votre fonction. Dans Lambda, un cas d'utilisation courant consiste à invoquer votre fonction en fonction d'un événement qui se produit ailleurs dans votre application. Certains services peuvent invoquer une fonction Lambda à chaque nouvel événement. C'est ce qu'on appelle un déclencheur. Pour les services basés sur les flux et les files d'attente, Lambda invoque la fonction avec des lots d'enregistrements. C'est ce qu'on appelle un mappage des sources d'événements.

Lorsque vous invoquez une fonction, vous pouvez choisir de le faire de façon synchrone ou asynchrone. Avec une invocation synchrone, vous attendez de la fonction qu'elle traite l'événement et renvoie une réponse. Avec l'invocation asynchrone, Lambda place en file d'attente l'événement à traiter, et renvoie une réponse immédiatement. Le paramètre de requête `InvocationType` de l'API Invoke détermine la manière dont Lambda appelle votre fonction. Une valeur `RequestResponse` indique une invocation synchrone et une valeur `Event` indique une invocation asynchrone.

Pour appeler votre fonction IPv6, utilisez les points de terminaison publics à double pile de Lambda. Les points de terminaison à double pile prennent en charge les deux et IPv4 . IPv6 Les points de terminaison à double pile de Lambda utilisent la syntaxe suivante :

```
protocol://lambda.us-east-1.api.aws
```

Vous pouvez également utiliser la fonction Lambda URLs pour appeler des fonctions. IPv6 Les points de terminaison URL de fonction ont le format suivant :

```
https://url-id.lambda-url.us-east-1.on.aws
```

Pour les invocations synchrones, si l'invocation de la fonction entraîne une erreur, consultez le message d'erreur dans la réponse et réessayez l'invocation manuellement. Pour les invocations asynchrones, Lambda gère les nouvelles tentatives et peut envoyer des enregistrements d'invocation à une destination.

Invocation d'une fonction Lambda de manière synchrone

Lorsque vous invoquez une fonction de façon synchrone, Lambda l'exécute et attend une réponse. Une fois l'exécution de la fonction terminée, Lambda renvoie la réponse du code de la fonction avec des données supplémentaires, telles que la version de la fonction qui a été invoquée. Pour invoquer une fonction de façon synchrone avec AWS CLI, utilisez la commande `invoke`.

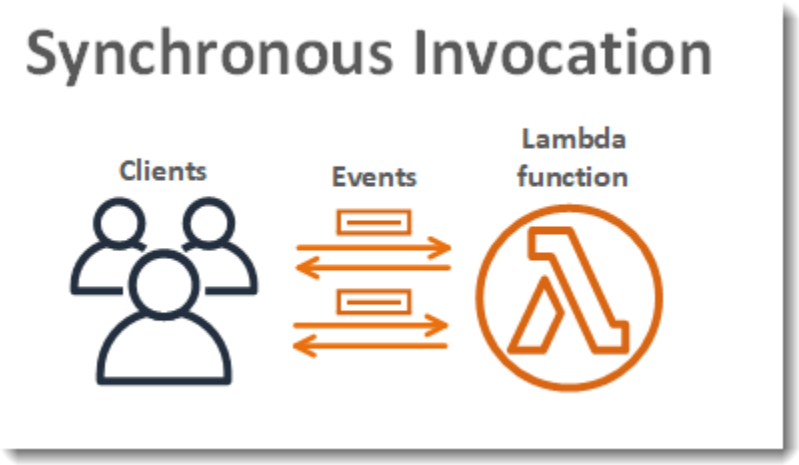
```
aws lambda invoke --function-name my-function \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "key": "value" }' response.json
```

L'option cli-binary-format est obligatoire si vous utilisez AWS CLI version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

Vous devriez voir la sortie suivante :

```
{
    "ExecutedVersion": "$LATEST",
    "StatusCode": 200
}
```

Le diagramme suivant montre des clients invoquant une fonction Lambda de manière synchrone. Lambda envoie les événements directement à la fonction et renvoie la réponse de la fonction à l'appelant.



`payload` est une chaîne qui contient un événement au format JSON. Le nom du fichier où l'AWS CLI écrit la réponse de la fonction est `response.json`. Si la fonction renvoie un objet ou une erreur, le corps de la réponse est l'objet ou l'erreur au format JSON. Si la fonction se termine sans erreur, le corps de la réponse est `null`.

> **Note**
>
> Lambda n'attend pas que les extensions externes se terminent avant d'envoyer la réponse. Les extensions externes s'exécutent comme des processus indépendants dans l'environnement d'exécution et continuent de s'exécuter après l'invocation de la fonction. Pour plus d'informations, consultez Utilisation des extensions Lambda pour augmenter vos fonctions Lambda.

La sortie de la commande qui s'affiche dans le terminal inclut des informations extraites d'en-têtes figurant dans la réponse de Lambda. Ces informations incluent la version qui a traité l'événement (utile lorsque vous utilisez des alias) et le code d'état renvoyé par Lambda. Si Lambda a pu exécuter la fonction, le code d'état est 200, même si la fonction a renvoyé une erreur.

> **Note**
>
> Pour les fonctions avec un long délai d'attente, votre client peut être déconnecté lors d'une invocation synchrone, pendant qu'il attend une réponse. Configurez votre client HTTP, SDK, pare-feu, proxy ou système d'exploitation pour permettre des connexions longues avec des paramètres de délai d'attente ou de keep-alive.

Si Lambda n'a pas pu exécuter la fonction, l'erreur s'affiche dans la sortie.

```
aws lambda invoke --function-name my-function \
    --cli-binary-format raw-in-base64-out \
    --payload value response.json
```

Vous devriez voir la sortie suivante:

```
An error occurred (InvalidRequestContentException) when calling the Invoke operation:
 Could not parse request body into json: Unrecognized token 'value': was expecting
 ('true', 'false' or 'null')
```

```
 at [Source: (byte[])"value"; line: 1, column: 11]
```

La AWS CLI est un outil à code open-source qui vous permet d'interagir avec les services AWS à l'aide de commandes dans votre ligne de commande shell. Pour effectuer les étapes de cette section, vous devez disposer de la version 2 de l'AWS CLI.

Vous pouvez utiliser AWS CLI pour récupérer les journaux d'une invocation à l'aide de l'option de commande `--log-type`. La réponse inclut un champ `LogResult` qui contient jusqu'à 4 Ko de journaux codés en base64 provenant de l'invocation.

Example récupérer un ID de journal

L'exemple suivant montre comment récupérer un ID de journal à partir du champ `LogResult` d'une fonction nommée `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Vous devriez voir la sortie suivante:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example décoder les journaux

Dans la même invite de commandes, utilisez l'utilitaire `base64` pour décoder les journaux. L'exemple suivant montre comment récupérer les journaux encodés en base64 pour `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

L'option cli-binary-format est obligatoire si vous utilisez AWS CLI version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

Vous devriez voir la sortie suivante :

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

L'utilitaire `base64` est disponible sous Linux, macOS et Ubuntu sous Windows. Les utilisateurs de macOS auront peut-être besoin d'utiliser `base64 -D`.

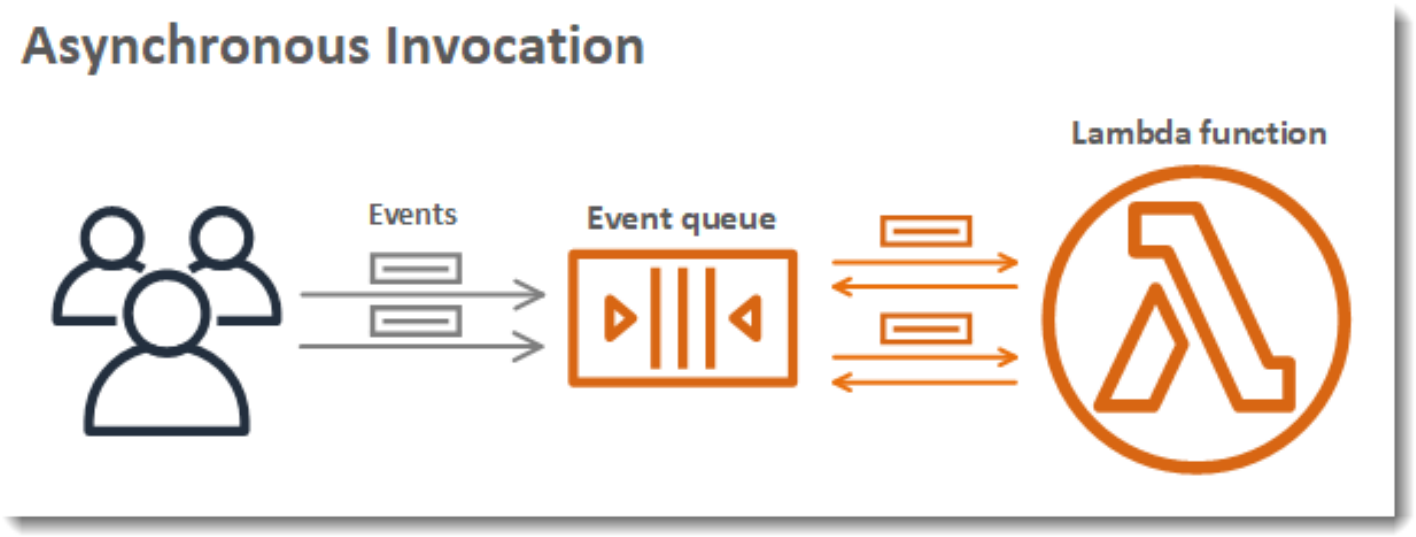
Pour plus d'informations sur l'API `Invoke` et la liste complète des paramètres, en-têtes et erreurs, consultez Invoquer.

Lorsque vous invoquez une fonction directement, vous pouvez vérifier la réponse pour les erreurs et réessayer. AWS CLI et le kit SDK AWS effectuent automatiquement de nouvelles tentatives selon les délais d'expiration du client, les limitations et les erreurs de service. Pour plus d'informations, consultez Présentation du comportement des nouvelles tentatives dans Lambda.

Invocation d'une fonction Lambda de manière asynchrone

Plusieurs services Services AWS, tels qu'Amazon Simple Storage Service (Amazon S3) et Amazon Simple Notification Service (Amazon SNS), invoquent des fonctions de manière asynchrone pour traiter les événements. Vous pouvez également invoquer une fonction Lambda de manière asynchrone à l'aide d'AWS Command Line Interface (AWS CLI) ou de l'un des kits SDK AWS. Lorsque vous invoquez une fonction de manière asynchrone, vous n'attendez pas de réponse de la part du code de la fonction. Vous remettez l'événement à Lambda qui se charge du reste. Vous pouvez configurer la façon dont Lambda gère les erreurs, et vous pouvez envoyer des enregistrements d'invocation à une ressource en aval telle qu'Amazon Simple Queue Service (Amazon SQS) ou Amazon EventBridge (EventBridge) pour enchaîner les composants de votre application.

Le diagramme suivant montre des clients invoquant une fonction Lambda de manière asynchrone. Lambda place les événements en file d'attente avant de les envoyer à la fonction.



Pour une invocation asynchrone, Lambda place l'événement dans une file d'attente et renvoie une réponse de succès sans plus informations. Un processus distinct lit les événements à partir de la file d'attente et les envoie à votre fonction.

Pour invoquer une fonction Lambda de manière asynchrone à l'aide de l'AWS Command Line Interface (AWS CLI) ou de l'un des kits SDK AWS, définissez le paramètre InvocationType sur `Event`. L'exemple suivant montre une commande AWS CLI permettant d'invoquer une fonction.

```
aws lambda invoke \
  --function-name my-function  \
```

```
  --invocation-type Event \
  --cli-binary-format raw-in-base64-out \
  --payload '{ "key": "value" }' response.json
```

Vous devriez voir la sortie suivante :

```
{
    "StatusCode": 202
}
```

L'option cli-binary-format est obligatoire si vous utilisez AWS CLI version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales prises en charge par l'AWS CLI dans le Guide de l'utilisateur AWS Command Line Interface version 2.

Le fichier de sortie (`response.json`) ne contient pas d'informations, mais il est tout de même créé lorsque vous exécutez cette commande. Si Lambda n'est pas en mesure d'ajouter l'événement à la file d'attente, un message d'erreur s'affiche dans la sortie de la commande.

Méthode de gestion des erreurs et nouvelles tentatives d'invocation asynchrone par Lambda

Lambda gère la file d'attente d'événements asynchrones de votre fonction, et effectue de nouvelles tentatives en cas d'erreur. Si la fonction renvoie une erreur, Lambda tente par défaut de l'exécuter deux fois de plus, avec une minute d'attente entre les deux premières tentatives, et deux minutes entre la deuxième et la troisième tentative. Les erreurs de fonction comprennent des erreurs renvoyées par le code de la fonction et des erreurs renvoyées par l'environnement d'exécution de la fonction, comme les délais d'expiration.

Si la fonction n'a pas suffisamment de simultanéité disponible pour traiter tous les événements, des demandes supplémentaires sont bloquées. Pour les erreurs de limitation (429) et les erreurs système (série 500), Lambda renvoie l'événement dans la file d'attente et tente d'exécuter à nouveau la fonction pendant une période pouvant aller jusqu'à 6 heures par défaut. L'intervalle de nouvelle tentative augmente de manière exponentielle de 1 seconde après la première tentative jusqu'à un maximum de 5 minutes. Si la file d'attente contient de nombreuses entrées, Lambda augmente l'intervalle entre les tentatives et réduit la fréquence des événements de lecture à partir de la file d'attente.

Même si votre fonction ne renvoie pas d'erreur, elle peut recevoir plusieurs fois le même événement de Lambda, car la file d'attente elle-même est cohérente à terme. Si la fonction ne peut pas à prendre en charge les événements entrants, ils peuvent également être supprimés de la file d'attente sans être envoyés à la fonction. Assurez-vous que le code de votre fonction gère correctement les événements dupliqués et que vous avez suffisamment de simultanéité disponible pour gérer toutes les invocations.

Lorsque la file d'attente est très longue, il peut arriver que de nouveaux événements expirent avant que Lambda ait la possibilité de les envoyer à votre fonction. Quand un événement expire ou échoue à toutes les tentatives de traitement, Lambda le supprime. Vous pouvez configurer la gestion des erreurs pour une fonction de façon à réduire le nombre de nouvelles tentatives que Lambda effectue, ou à supprimer les événements non traités plus rapidement. Pour capturer les événements supprimés, configurez une file d'attente de lettres mortes pour la fonction. Pour capturer les enregistrements des invocations ayant échoué (comme les expirations de délai d'expiration ou les erreurs d'environnement d'exécution), créez une destination en cas d'échec.

Configuration de la gestion des erreurs pour les invocations asynchrones Lambda

Utilisez les paramètres suivants pour configurer la façon dont Lambda gère les erreurs et les nouvelles tentatives pour les invocations de fonctions asynchrones :

- MaximumEventAgeInSeconds : la durée maximale, en secondes, pendant laquelle Lambda conserve un événement dans la file d'attente des événements asynchrones avant de le supprime.
- MaximumRetryAttempts : le nombre maximum de nouvelles tentatives que Lambda effectue quand la fonction renvoie une erreur.

Utilisez la console Lambda ou l'AWS CLI pour configurer les paramètres de gestion des erreurs sur une fonction, une version ou un alias.

Console

Pour configurer la gestion des erreurs

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Sélectionnez Configuration, puis Asynchronous invocation (Invocation asynchrone).

4. Sous Asynchronous invocation (Invocation asynchrone), choisissez Edit (Modifier).
5. Configurez les paramètres suivants.
   - Maximum age of event (Âge maximal de l'événement) – Durée maximale, jusqu'à 6 heures, pendant laquelle Lambda conserve un événement dans la file d'attente d'événements asynchrones.
   - Retry attempts (Nouvelles tentatives) – Nombre de nouvelles tentatives, entre 0 et 2, que Lambda effectue lorsque la fonction renvoie une erreur.
6. Choisissez Enregistrer.

AWS CLI

Pour configurer l'invocation asynchrone avec l'AWS CLI, utilisez la commande put-function-event-invoke-config. L'exemple suivant montre comment configurer une fonction avec un âge d'événement maximal de 1 heure et aucune nouvelle tentative.

```
aws lambda put-function-event-invoke-config \
  --function-name error \
  --maximum-event-age-in-seconds 3600 \
  --maximum-retry-attempts 0
```

La commande `put-function-event-invoke-config` remplace toute configuration existante sur la fonction, la version ou l'alias. Pour configurer une option sans en réinitialiser d'autres, utilisez update-function-event-invoke-config. L'exemple suivant illustre la configuration de Lambda pour l'envoie d'un enregistrement à une file d'attente SQS standard nommée `destination` lorsqu'un événement ne peut pas être traité.

```
aws lambda update-function-event-invoke-config \
  --function-name my-function \
  --destination-config '{"OnFailure":{"Destination": "arn:aws:sqs:us-
east-1:123456789012:destination"}}'
```

Vous devriez voir la sortie suivante :

```
{
    "LastModified": 1573686021.479,
    "FunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function:
$LATEST",
```

```
    "MaximumRetryAttempts": 0,
    "MaximumEventAgeInSeconds": 3600,
    "DestinationConfig": {
        "OnSuccess": {},
        "OnFailure": {}
    }
}
```

Quand un événement d'invocation dépasse l'âge maximum ou échoue à toutes les tentatives, Lambda le supprime. Pour conserver une copie des événements supprimés, configurez une destination pour les événements ayant échoué.

Capture des enregistrements d'invocations asynchrones Lambda

Lambda peut envoyer des enregistrements d'appels asynchrones à l'une des entités suivantes. Services AWS

- Amazon SQS : une file d'attente SQS standard
- Amazon SNS : une rubrique SNS standard
- Amazon S3 : un compartiment Amazon S3 (uniquement en cas d'échec)
- AWS Lambda : une fonction Lambda
- Amazon EventBridge — Un bus EventBridge événementiel

L'enregistrement d'invocation contient des détails sur la demande et la réponse au format JSON. Vous pouvez configurer des destinations distinctes pour les événements qui ont été traités avec succès et ceux dont le traitement a échoué. Vous pouvez également configurer une file d'attente Amazon SQS ou une rubrique Amazon SNS standard en tant que file d'attente de lettres mortes pour les événements abandonnés. Pour les files d'attente de lettres mortes, Lambda envoie uniquement le contenu de l'événement, sans plus d'informations sur la réponse.

Si Lambda ne parvient pas à envoyer un enregistrement vers une destination que vous avez configurée, il envoie une `DestinationDeliveryFailures` métrique à Amazon. CloudWatch Cela peut se produire si votre configuration inclut un type de destination non pris en charge, tel qu'une file d'attente FIFO Amazon SQS ou une rubrique FIFO Amazon SNS. Des erreurs de remise peuvent également se produire en raison d'erreurs d'autorisations et de limites de taille. Pour plus d'informations sur les métriques d'invocation Lambda, consultez the section called "Métriques d'invocation".

> **Note**
>
> Pour empêcher une fonction de se déclencher, vous pouvez définir la simultanéité réservée de la fonction sur zéro. Lorsque vous définissez la simultanéité réservée sur zéro pour une fonction invoquée de manière asynchrone, Lambda commence à envoyer les nouveaux événements à la file d'attente de lettres mortes ou à la destination d'événement en situation d'échec, sans aucune nouvelle tentative. Pour traiter les événements qui ont été envoyés alors que la simultanéité réservée était définie sur zéro, vous devez consommer les événements de la file d'attente de lettres mortes ou de la destination des événements en situation d'échec.

Ajout d'une destination

Pour retenir les enregistrements des invocations asynchrones, ajoutez une destination à votre fonction. Vous pouvez choisir d'envoyer les invocations réussies ou non à une destination. Chaque fonction peut avoir plusieurs destinations. Vous pouvez donc configurer des destinations distinctes pour les événements réussis et échoués. Chaque enregistrement envoyé à la destination est un document JSON avec des détails sur l'invocation. Comme les paramètres de gestion des erreurs, vous pouvez configurer des destinations au niveau d'une fonction, d'une version de fonction ou d'un alias.

> **Tip**
>
> Vous pouvez également conserver les enregistrements des appels ayant échoué pour les types de mappage de sources d'événements suivants : Amazon Kinesis, Amazon DynamoDB et Apache Kafka (Amazon MSK et Apache Kafka autogéré).

Le tableau suivant répertorie les destinations prises en charge pour les enregistrements d'invocation asynchrones. Pour que Lambda envoie correctement les enregistrements vers la destination que vous avez choisie, assurez-vous que le rôle d'exécution de votre fonction contient également les autorisations appropriées. Le tableau décrit également comment chaque type de destination reçoit l'enregistrement d'invocation JSON.

Type de destination	Autorisation obligatoire	Format JSON spécifique à la destination
File d'attente Amazon SQS	sqs : SendMessage	Lambda transmet l'enregis trement d'invocation en tant que `Message` à la destination.
Rubrique Amazon SNS	sns:Publish	Lambda transmet l'enregis trement d'invocation en tant que `Message` à la destination.
Compartiment Amazon S3 (uniquement en cas d'échec)	s3 : PutObject s3 : ListBucket	• Lambda stocke l'enregis trement d'invocation en tant qu'objet JSON dans le compartiment de destinati on. • Le nom de l'objet S3 utilise la convention de dénominat ion suivante : `aws/lambda/async/<function-name>/YYYY/MM/DD/YYYY-MM-DDTHH.MM.SS-<Random UUID>`
Fonction Lambda	lambda : InvokeFunction	Lambda transmet l'enregis trement d'invocation comme charge utile à la fonction.
EventBridge	événements : PutEvents	• Lambda transmet l'enregis trement d'invocation tel qu'il figure `detail` dans l'appel. PutEvents • La valeur du champ événement `source` est `aws.lambda` .

Type de destination	Autorisation obligatoire	Format JSON spécifique à la destination
		• La valeur du champ événement `detail-type` est soit Résultat de l'invocation de la fonction lambda – Succès, soit Résultat de l'invocation de la fonction lambda – Échec. • Le champ `resource` d'événement contient la fonction et la destination Amazon Resource Names (ARNs). • Pour les autres champs relatifs aux événements, consultez Amazon EventBridge events.

> **Note**
>
> Pour les destinations Amazon S3, si vous avez activé le chiffrement sur le compartiment à l'aide d'une clé KMS, votre fonction a également besoin de l'GenerateDataKeyautorisation kms :.

> **Important**
>
> Lorsque vous utilisez Amazon SNS comme destination, sachez que la taille maximale des messages d'Amazon SNS est limitée à 256 Ko. Si vos données utiles d'invocation asynchrone approchent 1 Mo, l'enregistrement d'invocation (qui inclut les données utiles d'origine ainsi que des métadonnées supplémentaires) peut dépasser la limite d'Amazon SNS et entraîner des échecs de livraison. Envisagez d'utiliser des destinations Amazon SQS ou Amazon S3 pour des données utiles plus importantes.

Les étapes suivantes expliquent comment configurer une destination pour une fonction à l'aide de la console Lambda et de l' AWS CLI.

Console

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Sous Function overview (Vue d'ensemble de la fonction), choisissez Add destination (Ajouter une destination).
4. Pour Source, choisissez Asynchronous invocation (Invocation asynchrone).
5. Pour Condition choisissez l'une des options suivantes :
   - On failure (En cas d'échec) – Envoyer un enregistrement quand l'événement échoue à toutes les tentatives de traitement ou dépasse l'âge maximal.
   - On success (En cas de succès) – Envoyer un enregistrement quand la fonction traite avec succès une invocation asynchrone.
6. Pour Type de destination, choisissez le type de ressource qui reçoit l'enregistrement d'invocation.
7. Pour Destination, choisissez une ressource.
8. Choisissez Enregistrer.

AWS CLI

Pour configurer une destination à l'aide de AWS CLI, exécutez la commande update-function-event-invoke-config. L'exemple suivant illustre la configuration de Lambda pour l'envoie d'un enregistrement à une file d'attente SQS standard nommée `destination` lorsqu'un événement ne peut pas être traité.

```
aws lambda update-function-event-invoke-config \
  --function-name my-function \
  --destination-config '{"OnFailure":{"Destination": "arn:aws:sqs:us-
east-1:123456789012:destination"}}'
```

Pratiques exemplaires en matière de sécurité pour les destinations Amazon S3

La suppression d'un compartiment S3 configuré comme destination sans supprimer la destination de la configuration de votre fonction peut engendrer un risque de sécurité. Si un autre utilisateur connaît le nom de votre compartiment de destination, il peut recréer le compartiment dans son Compte AWS. Les enregistrements des invocations ayant échoué seront envoyés dans son compartiment, exposant potentiellement les données de votre fonction.

> **Warning**
>
> Pour vous assurer que les enregistrements d'invocation de votre fonction ne peuvent pas être envoyés vers un compartiment S3 d'un autre Compte AWS, ajoutez une condition au rôle d'exécution de votre fonction qui limite `s3:PutObject` les autorisations aux compartiments de votre compte.

L'exemple suivant présente une politique IAM qui limite les autorisations `s3:PutObject` de votre fonction aux seuls compartiments de votre compte. Cette politique donne également à Lambda l'autorisation `s3:ListBucket` dont il a besoin pour utiliser un compartiment S3 comme destination.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "S3BucketResourceAccountWrite",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::*/*",
                "arn:aws:s3:::*"
            ],
            "Condition": {
                "StringEquals": {
                    "s3:ResourceAccount": "111122223333"
                }
            }
        }
    ]
```

```
}
```

Pour ajouter une politique d'autorisations au rôle d'exécution de votre fonction à l'aide du AWS Management Console ou AWS CLI, reportez-vous aux instructions des procédures suivantes :

Console

Pour ajouter une politique d'autorisations au rôle d'exécution d'une fonction (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction Lambda dont vous voulez modifier le rôle d'exécution.
3. Sous l'onglet Configuration, sélectionnez Autorisations.
4. Sous l'onglet Rôle d'exécution, sélectionnez le Nom du rôle de votre fonction pour ouvrir la page de console IAM du rôle.
5. Ajoutez une politique d'autorisations de au rôle en procédant comme suit :
   a. Dans le volet Politiques d'autorisations, choisissez Ajouter des autorisations, puis Créer une politique en ligne.
   b. Dans l'Éditeur de politique, sélectionnez JSON.
   c. Collez la politique que vous souhaitez ajouter dans l'éditeur (en remplacement du JSON existant), puis choisissez Suivant.
   d. Sous Détails de la politique, saisissez un Nom de la politique.
   e. Choisissez Create Policy (Créer une politique).

AWS CLI

Pour ajouter une politique d'autorisations au rôle d'exécution d'une fonction (CLI)

1. Créez un document de politique JSON avec les autorisations requises et enregistrez-le dans un répertoire local.
2. Utilisez la commande `put-role-policy` de la CLI IAM pour ajouter des autorisations au rôle d'exécution de votre fonction. Exécutez la commande suivante depuis le répertoire dans lequel vous avez enregistré votre document de politique JSON et remplacez le nom du rôle, le nom de la politique et le document de politique par vos propres valeurs.

```
aws iam put-role-policy \
--role-name my_lambda_role \
```

```
--policy-name LambdaS3DestinationPolicy \
--policy-document file://my_policy.json
```

Exemple d'enregistrement d'invocation

Quand une invocation correspond à la condition, Lambda envoie à la destination un document JSON avec des détails sur l'invocation. L'exemple suivant illustre un enregistrement d'invocation pour un événement dont le traitement a échoué à trois reprises en raison d'une erreur de fonction.

Example

```
{
    "version": "1.0",
    "timestamp": "2019-11-14T18:16:05.568Z",
    "requestContext": {
        "requestId": "e4b46cbf-b738-xmpl-8880-a18cdf61200e",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function:
$LATEST",
        "condition": "RetriesExhausted",
        "approximateInvokeCount": 3
    },
    "requestPayload": {
        "ORDER_IDS": [
            "9e07af03-ce31-4ff3-xmpl-36dce652cb4f",
            "637de236-e7b2-464e-xmpl-baf57f86bb53",
            "a81ddca6-2c35-45c7-xmpl-c3a03a31ed15"
        ]
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "responsePayload": {
        "errorMessage": "RequestId: e4b46cbf-b738-xmpl-8880-a18cdf61200e Process exited
 before completing request"
    }
}
```

L'enregistrement d'invocation contient des détails sur l'événement, la réponse et la raison pour laquelle l'enregistrement a été envoyé.

Suivi des demandes vers les destinations

Vous pouvez utiliser AWS X-Ray pour visualiser une vue connectée de chaque demande lorsqu'elle est placée en file d'attente, traitée par une fonction Lambda et transmise au service de destination. Lorsque vous activez le suivi X-Ray pour une fonction ou un service qui invoque une fonction, Lambda ajoute un en-tête X-Ray à la demande et transmet l'en-tête au service de destination. Les traces des services en amont sont automatiquement liées aux traces des fonctions Lambda en aval et des services de destination, créant ainsi une end-to-end vue de l'ensemble de l'application. Pour plus d'informations sur le suivi, consultez Visualisez les invocations de fonctions Lambda à l'aide de AWS X-Ray.

Ajout d'une file d'attente de lettres mortes

Comme alternative à une destination réservée aux échecs, vous pouvez configurer votre fonction avec une file d'attente de lettres mortes pour enregistrer les événements ignorés en vue d'un traitement ultérieur. Une file d'attente de lettres mortes agit de la même manière qu'une destination réservée aux échecs en ce sens qu'elle est utilisée lorsqu'un événement échoue à toutes les tentatives de traitement ou expire sans être traité. Toutefois, vous ne pouvez ajouter ou supprimer une file d'attente de lettres mortes qu'au niveau de la fonction. Les versions de fonctions utilisent les mêmes paramètres de file d'attente de lettres mortes que la version non publiée ($LATEST). Les destinations réservées aux échecs prennent également en charge des cibles supplémentaires et incluent des détails sur la réponse de la fonction dans l'enregistrement d'invocation.

Pour retraiter les événements d'une file d'attente de lettres mortes, vous pouvez la définir comme source d'événements pour votre fonction Lambda. Vous pouvez également récupérer manuellement les événements.

Vous pouvez choisir une file d'attente Amazon SQS standard ou une rubrique Amazon SNS standanrd pour votre file d'attente de lettres mortes. Les files d'attente FIFO et les rubriques FIFO Amazon SNS ne sont pas prises en charge.

- File d'attente Amazon SQS – Une file d'attente conserve les événements qui ont échoué jusqu'à ce qu'ils soient récupérés. Choisissez une file d'attente standard Amazon SQS si vous vous attendez à ce qu'une seule entité, telle qu'une fonction Lambda ou une CloudWatch alarme, traite l'événement défaillant. Pour de plus amples informations, veuillez consulter Utilisation de Lambda avec Amazon SQS.
- Rubrique Amazon SNS – Une rubrique relaie les événements qui ont échoué vers une ou plusieurs destinations. Choisissez une rubrique Amazon SNS standard si vous voulez que plusieurs entités agissent sur un événement ayant échoué. Par exemple, vous pouvez configurer une rubrique pour

envoyer des événements à une adresse e-mail, à une fonction Lambda ou à un point de and/or terminaison HTTP. Pour de plus amples informations, veuillez consulter Invocation des fonctions Lambda à l'aide des notifications Amazon SNS.

Pour envoyer des événements à une file d'attente ou une rubrique, votre fonction a besoin d'autorisations supplémentaires. Ajoutez une stratégie avec les autorisations requises au rôle d'exécution de votre fonction. Si la file d'attente ou le sujet cible est chiffré à l'aide d'une AWS KMS clé gérée par le client, assurez-vous que le rôle d'exécution de votre fonction et la politique basée sur les ressources de la clé contiennent les autorisations appropriées.

Après avoir créé la cible et mis à jour votre rôle d'exécution de la fonction, ajoutez la file d'attente de lettres mortes à votre fonction. Vous pouvez configurer plusieurs fonctions pour envoyer des événements à la même cible.

Console

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Sélectionnez Configuration, puis Asynchronous invocation (Invocation asynchrone).
4. Sous Asynchronous invocation (Invocation asynchrone), choisissez Edit (Modifier).
5. Définissez le service de file d'attente des lettres mortes sur Amazon SQS ou sur Amazon SNS.
6. Choisissez la file d'attente ou la rubrique cible.
7. Choisissez Enregistrer.

AWS CLI

Pour configurer une file d'attente contenant des lettres mortes avec le AWS CLI, utilisez la update-function-configurationcommande.

```
aws lambda update-function-configuration \
  --function-name my-function \
  --dead-letter-config TargetArn=arn:aws:sns:us-east-1:123456789012:my-topic
```

Lambda envoie l'événement à la file d'attente de lettres mortes en l'état, avec des informations supplémentaires dans les attributs. Vous pouvez utiliser ces informations pour identifier l'erreur que la

fonction a renvoyée ou établir une corrélation entre l'événement et des journaux ou une trace AWS X-Ray .

Attributs de message de file d'attente de lettres mortes

- RequestID (Chaîne) – ID de la demande d'invocation. IDsLes demandes apparaissent dans les journaux des fonctions. Vous pouvez également utiliser le kit SDK X-Ray pour enregistrer l'ID de demande sur un attribut dans le suivi. Vous pouvez ensuite rechercher des suivis par ID de demande dans la console X-Ray.
- ErrorCode(Numéro) — Le code d'état HTTP.
- ErrorMessage(String) — Les 1 premiers Ko du message d'erreur.

Si Lambda ne parvient pas à envoyer de message à la file d'attente contenant des lettres mortes, il supprime l'événement et émet la métrique. DeadLetterErrors Cela peut se produire en raison d'un manque d'autorisations ou si la taille totale du message est supérieure à la limite de la file d'attente cible ou rubrique. Supposons, par exemple, qu'une notification Amazon SNS dont le corps est proche de 1 Mo déclenche une fonction qui génère une erreur. Dans ce cas, le message peut dépasser la taille maximale autorisée dans la file d'attente de lettres mortes en raison des données d'événement ajoutées par Amazon SNS et de attributs ajoutés par Lambda.

Si vous utilisez Amazon SQS en tant que source d'événement, configurez une file d'attente de lettres mortes sur la file d'attente Amazon SQS elle-même, non sur la fonction Lambda. Pour de plus amples informations, veuillez consulter Utilisation de Lambda avec Amazon SQS.

Invoquer des fonctions Lambda durables

Les fonctions Lambda durables peuvent être invoquées en utilisant les mêmes méthodes que les fonctions Lambda par défaut, mais avec des considérations importantes pour les exécutions de longue durée. Cette section couvre les modèles d'invocation, la gestion de l'exécution et les meilleures pratiques pour des fonctions durables.

Limites d'invocation synchrone

Les appels synchrones de fonctions Lambda durables sont limités à 15 minutes, comme les fonctions Lambda par défaut. Si votre fonction durable doit s'exécuter pendant plus de 15 minutes, elle doit être invoquée de manière asynchrone.

Quand utiliser l'invocation synchrone : optez pour des fonctions durables qui s'exécutent en 15 minutes et lorsque vous avez besoin de résultats immédiats, telles que des flux de travail d'approbation rapides ou des tâches de traitement de données courtes.

Invocation asynchrone pour les flux de travail de longue durée

Pour les fonctions durables qui peuvent s'exécuter pendant plus de 15 minutes, utilisez l'invocation asynchrone. Cela permet à la fonction de continuer à fonctionner pendant que votre client reçoit un accusé de réception immédiat.

TypeScript

```
import { LambdaClient, InvokeCommand } from "@aws-sdk/client-lambda";

const client = new LambdaClient({});

// Asynchronous invocation
const command = new InvokeCommand({
  FunctionName: "my-durable-function",
  InvocationType: "Event", // Asynchronous
  Payload: JSON.stringify({ orderId: "12345" })
});

await client.send(command);
```

Python

```
import boto3
import json

client = boto3.client('lambda')

# Asynchronous invocation
response = client.invoke(
    FunctionName='my-durable-function',
    InvocationType='Event',  # Asynchronous
    Payload=json.dumps({'order_id': '12345'})
)
```

Gestion de l'exécution APIs

Lambda permet de gérer et APIs de surveiller les exécutions de fonctions durables, notamment en listant les exécutions, en obtenant le statut d'exécution et en arrêtant les exécutions en cours.

TypeScript

```
// Get execution status
const statusCommand = new InvokeCommand({
  FunctionName: "my-durable-function",
  InvocationType: "RequestResponse",
  Payload: JSON.stringify({
    action: "getStatus",
    executionId: "exec-123"
  })
});

const result = await client.send(statusCommand);
```

Python

```
# Get execution status
```

```
response = client.invoke(
    FunctionName='my-durable-function',
    InvocationType='RequestResponse',
    Payload=json.dumps({
        'action': 'get_status',
        'execution_id': 'exec-123'
    })
)
```

Procédure de traitement par Lambda des enregistrements provenant de sources d'événements basées sur des flux et des files d'attente

Un mappage des sources d'événements est une ressource Lambda qui lit des éléments à partir de services basés sur des flux et des files d'attente et qui invoque une fonction avec des lots d'enregistrements. Dans le cadre d'un mappage des sources d'événements, des ressources appelées sondeurs d'événements interrogent activement les nouveaux messages et invoquent des fonctions. Par défaut, Lambda adapte automatiquement les sondeurs d'événements, mais pour certains types de sources d'événements, vous pouvez utiliser le mode alloué pour contrôler le nombre minimum et maximum de sondeurs d'événements dédiés votre mappage des sources d'événements.

Les services suivants utilisent des mappages des sources d'événements pour invoquer des fonctions Lambda :

- Amazon DocumentDB (compatible avec MongoDB) (Amazon DocumentDB)
- Amazon DynamoDB
- Amazon Kinesis
- Amazon MQ
- Amazon Managed Streaming for Apache Kafka (Amazon MSK)
- Self-managed Apache Kafka
- Amazon Simple Queue Service (Amazon SQS)

> **Warning**
>
> Les mappages des sources d'événements Lambda traitent chaque événement au moins une fois, et le traitement des enregistrements peut être dupliqué. Pour éviter les problèmes potentiels liés à des événements dupliqués, nous vous recommandons vivement de rendre votre code de fonction idempotent. Pour en savoir plus, consultez Comment rendre ma fonction Lambda idempotente dans le Knowledge Center. AWS

Différence entre les mappages de sources d’événements et les déclencheurs directs

Certains Services AWS peuvent appeler directement des fonctions Lambda à l'aide de déclencheurs. Ces services envoient des événements à Lambda, et la fonction est invoquée aussitôt que l’événement spécifié se produit. Les déclencheurs sont adaptés aux événements discrets et au traitement en temps réel. Lorsque vous créez un déclencheur à l'aide de la console Lambda, celle-ci interagit avec le AWS service correspondant pour configurer la notification d'événement sur ce service. Le déclencheur est en fait stocké et géré par le service qui génère les événements, et non par Lambda. Voici quelques exemples de services qui utilisent des déclencheurs pour invoquer des fonctions Lambda :

- Amazon Simple Storage Service (Amazon S3) : invoque une fonction lorsqu’un objet est créé, supprimé ou modifié dans un compartiment. Pour de plus amples informations, veuillez consulter Didacticiel : utilisation d’un déclencheur Amazon S3 pour invoquer une fonction Lambda.
- Amazon Simple Notification Service (Amazon SNS) : invoque une fonction lorsqu’un message est publié dans une rubrique SNS. Pour de plus amples informations, veuillez consulter Tutoriel : Utilisation AWS Lambda avec Amazon Simple Notification Service.
- Amazon API Gateway : invoque une fonction lorsqu’une requête d’API est envoyée à un point de terminaison spécifique. Pour de plus amples informations, veuillez consulter Invocation d’une fonction Lambda à l’aide d’un point de terminaison Amazon API Gateway.

Les mappages des sources d’événements sont des ressources Lambda créées et gérées au sein du service Lambda. Les mappages des sources d’événements sont conçus pour traiter de gros volumes de données en streaming ou de messages provenant de files d’attente. Le traitement par lots des enregistrements d’un flux ou d’une file d’attente est plus efficace que le traitement individuel des enregistrements.

Comportement de traitement par lots

Par défaut, un mappage de source d’événements regroupe des enregistrements dans une même charge utile que Lambda envoie à votre fonction. Pour affiner le comportement du traitement par lots, vous pouvez configurer une fenêtre de traitement par lots (MaximumBatchingWindowInSeconds) et une taille de lot (BatchSize). Une fenêtre de traitement par lots représente l’intervalle de temps maximal pour collecter des enregistrements dans une même charge utile. La taille d’un lot est le

nombre maximal d'enregistrements dans un même lot. Lambda invoque votre fonction en présence de l'un des trois critères suivants :

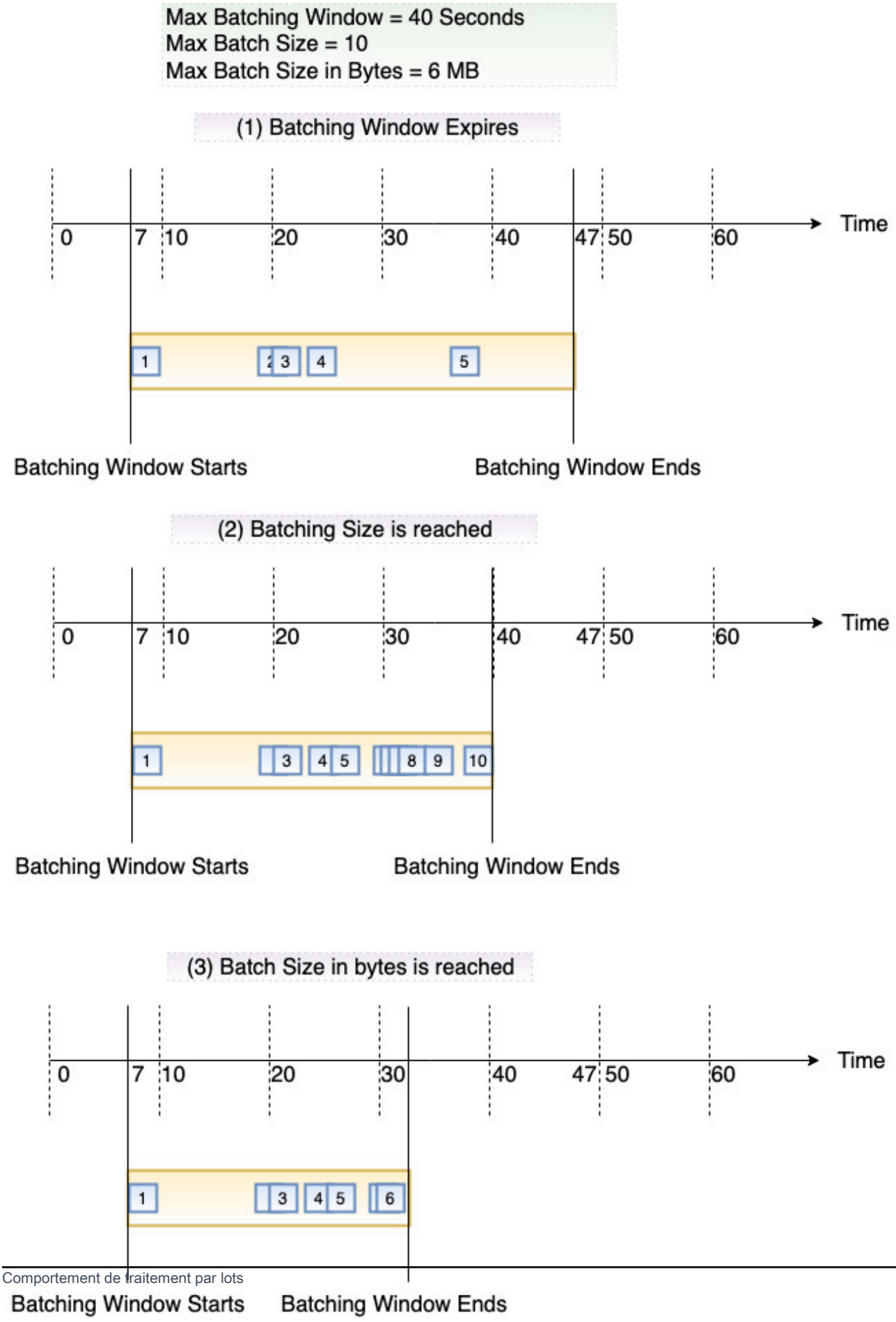
- La fenêtre de traitement par lots atteint sa valeur maximale. Le comportement par défaut de la fenêtre de traitement par lots varie selon la source d'événement spécifique.
  - Pour les sources d'événements Kinesis, DynamoDB et Amazon SQS : la fenêtre de traitement par lot par défaut est de 0 seconde. Cela signifie que Lambda invoque votre fonction dès que des enregistrements sont disponibles. Pour définir une fenêtre de traitement par lots, configurez `MaximumBatchingWindowInSeconds`. Vous pouvez configurer ce paramètre à n'importe quelle valeur comprise entre 0 et 300 secondes par incréments d'1 seconde. Si vous configurez une fenêtre de traitement par lots, la fenêtre suivante commence dès que l'invocation de fonction précédente est terminée.
  - Pour les sources d'événements Amazon MSK, Apache Kafka autogérées, Amazon MQ et Amazon DocumentDB : la fenêtre de traitement par lots par défaut est de 500 ms. Vous pouvez configurer `MaximumBatchingWindowInSeconds` à n'importe quelle valeur comprise entre 0 et 300 secondes par incréments de secondes. En mode provisionné pour les mappages des sources d'événements Kafka, lorsque vous configurez une fenêtre de traitement par lots, la fenêtre suivante commence dès que le lot précédent est terminé. Dans les mappages des sources d'événements Kafka non provisionnés, si vous configurez une fenêtre de traitement par lots, la fenêtre suivante commence dès que l'invocation de fonction précédente est terminée. Pour minimiser la latence lors de l'utilisation des mappages des sources d'événements Kafka en mode provisionné, définissez `MaximumBatchingWindowInSeconds` sur 0. Ce paramètre garantit que Lambda commencera à traiter le lot suivant immédiatement après avoir terminé l'invocation de fonction en cours. Pour plus d'informations sur le traitement à faible latence, consultez Apache Kafka à faible latence.
  - Pour les sources d'événements Amazon MQ et Amazon DocumentDB : la fenêtre de traitement par lots par défaut est de 500 ms. Vous pouvez configurer `MaximumBatchingWindowInSeconds` à n'importe quelle valeur comprise entre 0 et 300 secondes par incréments de secondes. Une fenêtre de traitement par lots commence dès l'arrivée du premier registre.

> **Note**
>
> Étant donné que vous ne pouvez modifier `MaximumBatchingWindowInSeconds` que par incréments de secondes, vous ne pouvez pas revenir à la fenêtre de traitement par

lots par défaut de 500 ms après l'avoir modifiée. Pour restaurer la fenêtre de traitement par lots par défaut, vous devez créer un mappage de source d'événement.

- La taille du lot est atteinte. La taille minimale du lot est de 1. La taille par défaut et la taille maximale du lot dépendent de la source d'événement. Pour plus d'informations sur ces valeurs, consultez la spécification BatchSize pour l'opération de l'API `CreateEventSourceMapping`.
- La taille de la charge utile atteint 6 Mo. Vous ne pouvez pas modifier cette limite.

Le diagramme suivant illustre ces trois conditions. Supposons qu'une fenêtre de traitement par lots commence à `t = 7` secondes. Dans le premier scénario, la fenêtre de traitement par lots atteint son maximum de 40 secondes à `t = 47` secondes après avoir accumulé 5 enregistrements. Dans le second scénario, la taille du lot atteint 10 avant l'expiration de la fenêtre de traitement par lots, de sorte que la fenêtre de traitement par lots se termine plus tôt. Dans le troisième scénario, la taille maximale de la charge utile est atteinte avant l'expiration de la fenêtre de traitement par lots, de sorte que la fenêtre de traitement par lots se termine plus tôt.


Max Batching Window = 40 Seconds
Max Batch Size = 10
Max Batch Size in Bytes = 6 MB
(1) Batching Window Expires
0
7
10
20
30
40
47
50
60
Time
1
3
4
5
Batching Window Starts
Batching Window Ends
(2) Batching Size is reached
0
7
10
20
30
40
47
50
60
Time
1
3
4
5
8
9
10
Batching Window Starts
Batching Window Ends
(3) Batch Size in bytes is reached
0
7
10
20
30
40
47
50
60
Time
1
3
4
5
6
Batching Window Starts
Batching Window Ends

Nous vous recommandons de tester différentes tailles de lots et d'enregistrements afin que la fréquence d'interrogation de chaque source d'événement soit adaptée à la rapidité avec laquelle votre fonction est en mesure d'accomplir sa tâche. Le CreateEventSourceMapping`BatchSize`paramètre contrôle le nombre maximum d'enregistrements qui peuvent être envoyés à votre fonction à chaque appel. Une taille de lot plus grande peut souvent absorber plus efficacement l'invocation sur un plus large ensemble d'enregistrements, ce qui accroît votre débit.

Lambda envoi le prochain lot pour traitement sans attendre que les extensions configurées soient terminées. En d'autres termes, vos extensions peuvent continuer à s'exécuter pendant que Lambda traite le prochain lot d'enregistrements. Cela peut causer des problèmes de limitation si vous enfreignez l'un des paramètres ou l'une des limites de simultanéité de votre compte. Pour détecter s'il s'agit d'un problème éventuel, surveillez vos fonctions et vérifiez si vous observez des métriques de simultanéité plus élevées que prévu pour votre mappage des sources d'événements. En raison de la brièveté des intervalles entre les invocations, Lambda peut brièvement signaler une utilisation simultanée supérieure au nombre de partitions. Cela peut être vrai même pour les fonctions Lambda sans extensions.

Par défaut, si votre fonction renvoie une erreur, le mappage de la source d'événements retraite l'ensemble du lot jusqu'à ce que la fonction réussisse ou que les éléments du lot arrivent à expiration. Pour s'assurer d'un traitement dans l'ordre, le mappage de la source d'événement met en pause le traitement dans la partition concernée jusqu'à la résolution de l'erreur. Pour les sources de flux (DynamoDB et Kinesis), vous pouvez configurer le nombre maximal de tentatives que Lambda effectue lorsque votre fonction renvoie une erreur. Les erreurs de service ou les limitations où le lot n'atteint pas votre fonction ne comptent pas dans les tentatives de réessai. Vous pouvez également configurer le mappage des sources d'événements pour qu'il envoie un enregistrement d'invocation à une destination lorsqu'il rejette un lot d'événements.

Mode alloué

Les mappages des sources d'événements Lambda utilisent des sondeurs d'événements pour interroger votre source d'événements à la recherche de nouveaux messages. Par défaut, Lambda gère le dimensionnement automatique de ces sondeurs en fonction du volume des messages. Lorsque le trafic de messages augmente, Lambda augmente automatiquement le nombre de sondeurs d'événements pour gérer la charge, et le réduit lorsque le trafic diminue.

En mode provisionné, vous pouvez affiner le débit du mappage de vos sources d'événements en définissant des limites minimales et maximales pour les ressources de sondage dédiées qui restent

prêtes à gérer les modèles de trafic attendus. Ces ressources évoluent automatiquement 3 fois plus vite pour faire face aux pics soudains de trafic d'événements et fournissent une capacité 16 fois supérieure pour traiter des millions d'événements. Cela vous permet de créer des charges de travail hautement réactives axées sur les événements avec des exigences de performance strictes.

Dans Lambda, un sondeur d'événements est une unité de calcul dont les capacités de débit varient selon le type de source d'événements. Pour Amazon MSK et Apache Kafka autogéré, chaque sondeur d'événements peut gérer jusqu'à 5 % MB/sec du débit ou jusqu'à 5 appels simultanés. Par exemple, si la source de votre événement produit une charge utile moyenne de 1 Mo et que la durée moyenne de votre fonction est de 1 seconde, un seul sondeur d'événements Kafka peut prendre en charge 5 MB/sec débits et 5 appels Lambda simultanés (en supposant qu'il n'y ait aucune transformation de charge utile). Pour Amazon SQS, chaque sondeur d'événements peut gérer jusqu'à 1 % MB/sec du débit ou jusqu'à 10 appels simultanés. L'utilisation du mode provisionné entraîne des coûts supplémentaires en fonction de l'utilisation de votre sondeur d'événements. Pour plus d’informations sur la tarification, consultez Tarification AWS Lambda.

Le mode provisionné est disponible pour les sources d'événements Amazon MSK, Apache Kafka autogérées et Amazon SQS. Alors que les paramètres de simultanéité vous permettent de contrôler la mise à l’échelle de votre fonction, le mode alloué vous permet de contrôler le débit de votre mappage des sources d’événements. Pour garantir des performances optimales, vous devrez peut-être ajuster les deux paramètres indépendamment.

Le mode provisionné est idéal pour les applications en temps réel nécessitant une latence constante pour le traitement des événements, telles que les sociétés de services financiers traitant les flux de données du marché, les plateformes de commerce électronique fournissant des recommandations personnalisées en temps réel et les sociétés de jeux gérant les interactions avec les joueurs en direct.

Chaque sondeur d'événements prend en charge une capacité de débit différente :

- Pour Amazon MSK et Apache Kafka autogéré : jusqu'à 5 % MB/sec du débit ou jusqu'à 5 appels simultanés
- Pour Amazon SQS : jusqu'à 1 % MB/sec de débit ou jusqu'à 10 appels simultanés ou jusqu'à 10 appels d'API d'interrogation SQS par seconde.

Pour les mappages de sources d'événements Amazon SQS, vous pouvez définir le nombre minimum de sondeurs entre 2 et 200 avec une valeur par défaut de 2, et le nombre maximum entre 2 et 2 000 avec une valeur par défaut de 200. Lambda adapte le nombre de sondeurs d'événements entre le

minimum et le maximum configurés, en ajoutant rapidement jusqu'à 1 000 appels simultanés par minute afin de fournir un traitement cohérent et à faible latence de vos événements.

Pour les mappages de sources d'événements Kafka, vous pouvez définir le nombre minimum de sondeurs entre 1 et 200 avec une valeur par défaut de 1, et le nombre maximum entre 1 et 2 000 avec une valeur par défaut de 200. Lambda adapte le nombre de sondeurs d'événements entre le minimum et le maximum configurés en fonction de votre historique d'événements dans votre rubrique afin de garantir un traitement à faible latence de vos événements.

Notez que pour les sources d'événements Amazon SQS, le paramètre de simultanéité maximale ne peut pas être utilisé en mode provisionné. Lorsque vous utilisez le mode provisionné, vous contrôlez la simultanéité par le biais du paramètre Nombre maximal de sondeurs d'événements.

Pour plus d’informations sur la configuration du mode alloué, reportez-vous aux sections suivantes :

- Configuration du mode alloué pour les mappages des sources d’événements Amazon MSK
- Configuration du mode alloué pour les mappages des source d’événement Apache Kafka autogéré
- Utilisation du mode provisionné avec les mappages de sources d'événements Amazon SQS

Pour minimiser le temps de latence en mode provisionné, définissez ce paramètre `MaximumBatchingWindowInSeconds` sur 0. Ce paramètre garantit que Lambda commencera à traiter le lot suivant immédiatement après avoir terminé l’invocation de fonction en cours. Pour plus d’informations sur le traitement à faible latence, consultez Apache Kafka à faible latence.

Après avoir configuré le mode alloué, vous pouvez observer l’utilisation des sondeurs d’événements pour votre charge de travail en surveillant la métrique `ProvisionedPollers`. Pour de plus amples informations, veuillez consulter the section called “Métriques de mappage des sources d’événements”.

API de mappage de la source d’événement

Pour gérer une source d’événement à l’aide de la AWS Command Line Interface (AWS CLI) ou d’un AWS SDK, vous pouvez utiliser les opérations d’API suivantes :

- CreateEventSourceMapping
- ListEventSourceMappings
- GetEventSourceMapping
- UpdateEventSourceMapping

- DeleteEventSourceMapping

Utilisation des balises dans les mappages des sources d’événements

Vous pouvez étiqueter les mappages des sources d’événements pour organiser et gérer vos ressources. Les balises sont des paires clé-valeur de forme libre associées à vos ressources prises en charge par l’ensemble des ressources Services AWS. Pour plus d'informations sur les cas d'utilisation des balises, consultez la section Stratégies de balisage courantes dans le guide des AWS ressources de balisage et de l'éditeur de balises.

Les mappages des sources d’événements sont associés à des fonctions, qui peuvent avoir leurs propres balises. Les mappages des sources d’événements n’héritent pas automatiquement des balises des fonctions. Vous pouvez utiliser l' AWS Lambda API pour afficher et mettre à jour les balises. Vous pouvez également consulter et mettre à jour les balises tout en gérant un mappage de source d'événement spécifique dans la console Lambda, y compris ceux utilisant le mode provisionné pour Amazon SQS.

Autorisations requises pour l’utilisation des balises

Pour autoriser une identité Gestion des identités et des accès AWS (IAM) – utilisateur, groupe ou rôle – à afficher ou marquer les ressources, accordez-lui les autorisations correspondantes :

- lambda : ListTags —Lorsqu' une ressource possède des balises, accordez cette autorisation à tous ceux qui ont besoin de `ListTags` l'utiliser. Pour les fonctions balisées, cette autorisation est également nécessaire pour `GetFunction`.
- lambda : TagResource —Accordez cette autorisation à toute personne ayant besoin d'appeler `TagResource` ou d'exécuter un tag lors de la création.

Vous pouvez éventuellement envisager d'accorder également l'UntagResourceautorisation lambda : pour autoriser les `UntagResource` appels à la ressource.

Pour de plus amples informations, veuillez consulter Politiques IAM basées sur l’identité pour Lambda.

Utilisation des balises avec la console Lambda

Vous pouvez utiliser la console Lambda pour créer des mappages de sources d'événements comportant des balises, ajouter des balises aux mappages de sources d'événements existants

et filtrer les mappages de sources d'événements par balise, y compris ceux configurés en mode provisionné pour Amazon SQS.

Lorsque vous ajoutez un déclencheur pour les services basés sur les flux et les files d'attente pris en charge à l'aide de la console Lambda, Lambda crée automatiquement un mappage des sources d'événements. Pour plus d'informations sur ces sources d'événements, consultez the section called "Mappages des sources d'événements". Pour créer un mappage des sources d'événement dans la console, vous devez disposer des éléments suivants :

- Une fonction .
- Une source d'événement provenant d'un service concerné.

Vous pouvez ajouter les balises dans le cadre de l'interface utilisateur que vous utilisez pour créer ou mettre à jour des déclencheurs.

Pour ajouter une balise lors de la création d'un mappage des sources d'événements

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom de votre fonction .
3. Sous Function overview (Vue d'ensemble de la fonction), choisissez Add trigger (Ajouter un déclencheur).
4. Sous Configuration du déclencheur, dans la liste déroulante, choisissez le nom du service dont provient votre source d'événement.
5. Fournissez la configuration de base de votre source d'événements. Pour plus d'informations sur la configuration de votre source d'événements, consultez la section relative au service correspondant dans Intégration d'autres services.
6. Sous Configuration du mappage des sources d'événements, sélectionnez Paramètres supplémentaires.
7. Sous Balises, choisissez Ajouter une nouvelle balise.
8. Dans le champ Clé, saisissez la clé de votre balise. Pour plus d'informations sur les restrictions relatives au balisage, consultez la section Limites et exigences relatives au nommage des balises dans le Guide des AWS ressources de balisage et de l'éditeur de balises.
9. Choisissez Ajouter.

Pour ajouter des balises à un mappage des sources d'événements existant

1. Ouvrez la page Mappages des sources d'événements dans la console Lambda.
2. Dans la liste des ressources, choisissez l'UUID du mappage des sources d'événements correspondant à l'ARN de la source d'événement et de la fonction.
3. Dans la liste des onglets située sous le volet de configuration générale, choisissez Balises.
4. Choisissez Gérer les balises.
5. Choisissez Add new tag (Ajouter une nouvelle balise).
6. Dans le champ Clé, saisissez la clé de votre balise. Pour plus d'informations sur les restrictions relatives au balisage, consultez la section Limites et exigences relatives au nommage des balises dans le Guide des AWS ressources de balisage et de l'éditeur de balises.
7. Choisissez Enregistrer.

Pour filtrer les mappages des sources d'événements par balise

1. Ouvrez la page Mappages des sources d'événements dans la console Lambda.
2. Cliquez sur la barre de recherche.
3. Dans la liste déroulante, sélectionnez votre clé de balise sous le sous-titre Balises.
4. Sélectionnez Utiliser : « tag-name » pour afficher tous les mappages des sources d'événements étiquetés avec cette touche, ou choisissez un Opérateur pour affiner le filtrage en fonction de la valeur.
5. Sélectionnez votre valeur de balise pour appliquer un filtre combinant la clé et la valeur de la balise.

La zone de recherche prend également en charge la recherche de clés de balise. Saisissez le nom d'une clé pour la rechercher dans la liste.

Utilisation de balises avec AWS CLI

Vous pouvez ajouter et supprimer des balises sur les ressources Lambda existantes, mappages de sources d'événements inclus, avec l'API Lambda. Il est également possible d'ajouter des balises lors de la création d'un mappage des sources d'événements, vous permettant ainsi de conserver une ressource balisée tout au long de son cycle de vie.

Mise à jour des balises avec la balise Lambda APIs

Vous pouvez ajouter et supprimer des balises pour les ressources Lambda prises en charge par le biais des opérations TagResourceet de l'UntagResourceAPI.

Vous pouvez appeler ces opérations par l'intermédiaire de l' AWS CLI. Pour ajouter des balises à une ressource existante, utilisez la commande `tag-resource`. Cet exemple ajoute deux balises, l'une avec la clé *`Department`* et l'autre avec la clé*`CostCenter`*.

```
aws lambda tag-resource \
--resource arn:aws:lambda:us-east-2:123456789012:resource-type:my-resource \
--tags Department=Marketing,CostCenter=1234ABCD
```

Pour supprimer des balises, utilisez la commande `untag-resource`. Cet exemple supprime la balise contenant la clé*`Department`*.

```
aws lambda untag-resource --resource arn:aws:lambda:us-east-1:123456789012:resource-
type:resource-identifier \
--tag-keys Department
```

Ajout de balises lors de la création d'un mappage des sources d'événements

Pour créer un nouveau mappage de source d'événements Lambda avec des balises, utilisez l'opération CreateEventSourceMappingAPI. Spécifiez le paramètre `Tags`. Vous pouvez appeler cette opération à l'aide de la `create-event-source-mapping` AWS CLI commande et de l'`--tags`option. Pour plus d'informations sur la commande CLI, consultez create-event-source-mappingla référence des AWS CLI commandes.

Avant d'utiliser le paramètre `Tags` avec `CreateEventSourceMapping`, vérifiez que votre rôle dispose de l'autorisation d'étiqueter les ressources en plus des autorisations habituelles nécessaires à cette opération. Pour plus d'informations sur les autorisations requises pour l'étiquetage, consultez the section called "Autorisations requises pour l'utilisation des balises".

Afficher les tags avec le tag Lambda APIs

Pour afficher les balises associées à une ressource Lambda spécifique, utilisez l'opération d'API `ListTags`. Pour de plus amples informations, veuillez consulter ListTags.

Vous pouvez appeler cette opération à l'aide de la `list-tags` AWS CLI commande en fournissant un ARN (Amazon Resource Name).

```
aws lambda list-tags --resource arn:aws:lambda:us-east-1:123456789012:resource-type:resource-identifier
```

Filtrage de ressources par balise

Vous pouvez utiliser le fonctionnement de l' AWS Resource Groups Tagging API GetResourcesAPI pour filtrer vos ressources par balises. L'opération `GetResources` reçoit jusqu'à 10 filtres, chaque filtre contenant une clé de balise et jusqu'à 10 valeurs de balise. Vous fournissez `GetResources`avec un `ResourceType` pour filtrer par certains types de ressources.

Vous pouvez appeler cette opération à l'aide de la `get-resources` AWS CLI commande. Pour des exemples d'utilisation de `get-resources`, consultez get-resources dans la Référence des commandes de l'AWS CLI.

Contrôle des événements envoyés par Lambda à votre fonction

Vous pouvez utiliser le filtrage d'événements pour contrôler les enregistrements d'un flux ou d'une file d'attente que Lambda envoie à votre fonction. Par exemple, vous pouvez ajouter un filtre pour que votre fonction ne traite que les messages Amazon SQS contenant certains paramètres de données. Le filtrage des sources d'événements ne fonctionne qu'avec certains mappages des sources d'événements. Vous pouvez ajouter des filtres aux mappages de sources d'événements pour les éléments suivants : Services AWS

- Amazon DynamoDB
- Amazon Kinesis Data Streams
- Amazon MQ
- Amazon Managed Streaming for Apache Kafka (Amazon MSK)
- Self-managed Apache Kafka
- Amazon Simple Queue Service (Amazon SQS)

Pour obtenir des informations spécifiques sur le filtrage avec des sources d'événements précises, consultez the section called "Utilisation de filtres avec différents Services AWS". Lambda ne prend pas en charge le filtrage d'événements pour Amazon DocumentDB.

Par défaut, vous pouvez définir jusqu'à cinq filtres différents pour un seul mappage des sources d'événements. Vos filtres sont logiquement ORed combinés. Si un enregistrement de votre source d'événement satisfait un ou plusieurs de vos filtres, Lambda inclut l'enregistrement dans le prochain événement qu'il envoie à votre fonction. Si aucun de vos filtres n'est satisfait, Lambda rejette l'enregistrement.

> **Note**
>
> Si vous avez besoin de définir plus de cinq filtres pour une source d'événement, vous pouvez demander une augmentation de quota jusqu'à dix filtres pour chaque source d'événement. Si vous tentez d'ajouter plus de filtres que votre quota actuel ne le permet, Lambda renvoie une erreur lorsque vous essayez de créer la source d'événement.

Rubriques

- Présentation des principes de base du filtrage d'événements

- Traitement des enregistrements ne répondant pas aux critères de filtrage
- Syntaxe des règles de filtrage
- Attacher des critères de filtre à un mappage de sources d'événements (console)
- Attacher des critères de filtre à un mappage de sources d'événements (AWS CLI)
- Attacher des critères de filtre à un mappage de sources d'événements (AWS SAM)
- Chiffrement des critères de filtre
- Utilisation de filtres avec différents Services AWS

Présentation des principes de base du filtrage d'événements

Un objet (`FilterCriteria`) de critères de filtre est une structure composée d'une liste de filtres (`Filters`). Chaque filtre est une structure qui définit un modèle de filtrage d'événements (`Pattern`). Un modèle est une représentation sous forme de chaîne d'une règle de filtre JSON. La structure d'un objet `FilterCriteria` est la suivante.

```
{
   "Filters": [
        {
            "Pattern": "{ \"Metadata1\": [ rule1 ], \"data\": { \"Data1\":
 [ rule2 ] }}"
        }
    ]
}
```

Pour plus de clarté, voici la valeur du `Pattern` de filtre étendu en JSON simple :

```
{
    "Metadata1": [ rule1 ],
    "data": {
        "Data1": [ rule2 ]
    }
}
```

Votre modèle de filtrage peut inclure des propriétés de métadonnées, des propriétés de données ou les deux. Les paramètres de métadonnées disponibles et le format des paramètres de données varient en fonction de l'objet Service AWS qui sert de source d'événements. Par exemple, supposons que votre mappage des sources d'événements reçoit l'enregistrement suivant d'une file d'attente Amazon SQS :

```
{
    "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
    "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
    "body": "{\\n \"City\": \"Seattle\",\\n \"State\": \"WA\",\\n \"Temperature\":
 \"46\"\\n}",
    "attributes": {
        "ApproximateReceiveCount": "1",
        "SentTimestamp": "1545082649183",
        "SenderId": "AIDAIENQZJOLO23YVJ4VO",
        "ApproximateFirstReceiveTimestamp": "1545082649185"
    },
    "messageAttributes": {},
    "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
    "eventSource": "aws:sqs",
    "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
    "awsRegion": "us-east-2"
}
```

- Les propriétés de métadonnées sont les champs contenant des informations sur l'événement qui a créé l'enregistrement. Dans l'exemple d'enregistrement Amazon SQS, les propriétés de métadonnées comprennent des champs tels que `messageID`, `eventSourceArn` et `awsRegion`.
- Les propriétés de données sont les champs de l'enregistrement contenant les données de votre flux ou file d'attente. Dans l'exemple d'événement Amazon SQS, la clé du champ de données est `body` et les propriétés de données sont les champs `City` `State` et `Temperature`.

Les différents types de sources d'événements utilisent des valeurs clés différentes pour leurs champs de données. Pour filtrer sur les propriétés de données, assurez-vous d'utiliser la bonne clé dans le modèle de votre filtre. Pour obtenir la liste des clés de filtrage des données et des exemples de modèles de filtre pris en charge pour chacune d'entre elles Service AWS, reportez-vous àUtilisation de filtres avec différents Services AWS.

Le filtrage d'événements peut gérer le filtrage JSON à plusieurs niveaux. Par exemple, considérez le fragment suivant d'un enregistrement provenant d'un flux DynamoDB :

```
"dynamodb": {
    "Keys": {
        "ID": {
            "S": "ABCD"
        }
        "Number": {
```

```
            "N": "1234"
    },
    ...
}
```

Supposons que vous vouliez traiter uniquement les enregistrements dont la valeur de la clé de tri `Number` est 4 567. Dans ce cas, votre objet `FilterCriteria` se présente comme suit :

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\": { \"Keys\": { \"Number\": { \"N\":
 [ "4567" ] } } } }"
        }
    ]
}
```

Pour plus de clarté, voici la valeur du `Pattern` de filtre étendu en JSON simple :

```
{
    "dynamodb": {
        "Keys": {
            "Number": {
                "N": [ "4567" ]
                }
            }
        }
}
```

Traitement des enregistrements ne répondant pas aux critères de filtrage

La manière dont Lambda gère les enregistrements qui ne répondent pas à vos critères de filtre dépend de la source de l'événement.

- Pour Amazon SQS, si un message ne répond pas à vos critères de filtre, Lambda supprime automatiquement le message de la file d'attente. Vous n'avez pas besoin de supprimer manuellement ces messages dans Amazon SQS.
- Pour Kineses et DynamoDB, après que vos critères de filtre ont évalué un enregistrement, l'itérateur de flux avance au-delà de cet enregistrement. Si l'enregistrement ne répond pas à vos critères de filtre, vous n'avez pas besoin de supprimer manuellement l'enregistrement de la

source de votre événement. Après la période de conservation, Kinesis et DynamoDB suppriment automatiquement ces anciens enregistrements. Si vous souhaitez que les enregistrements soient supprimés plus tôt, consultez Modification de la période de conservation des données.

- Pour les messages Amazon MSK, Apache Kafka autogéré et Amazon MQ, Lambda supprime les messages qui ne correspondent pas à tous les champs inclus dans le filtre. Pour Amazon MSK et Apache Kafka autogéré, Lambda valide les décalages pour les messages correspondants et non correspondants après avoir invoqué la fonction avec succès. Pour Amazon MQ, Lambda accuse réception des messages correspondants après avoir correctement invoqué la fonction et reconnaît les messages non correspondants lors du filtrage.

Syntaxe des règles de filtrage

Pour les règles de filtrage, Lambda prend en charge EventBridge les règles Amazon et utilise la même syntaxe que. EventBridge Pour plus d'informations, consultez les modèles EventBridge d'événements Amazon dans le guide de EventBridge l'utilisateur Amazon.

Voici un récapitulatif de tous les opérateurs de comparaison disponibles pour le filtrage d'événement Lambda.

Opérateur de comparaison	Exemple	Syntaxe des règles
Null	UserID est null	"UserID": [null]
Vide	LastName est vide	"LastName": [""]
Égal à	Le nom est « Alice »	"Name": ["Alice"]
Est égal à (ignorer les majuscules)	Le nom est « Alice »	« Nom » : [{"equals-ignore-case« : « alice »}]
Et	Le lieu est « New York » et le jour est « Monday »	"Location": ["New York"], "Day": ["Monday"]
Or	PaymentType est « Crédit » ou « Débit »	PaymentType« : [« Crédit », « Débit"]

Opérateur de comparaison	Exemple	Syntaxe des règles
Ou (plusieurs champs)	Location est « New York », ou Day est « Monday ».	"$or": [{ "Location": ["New York"] }, { "Day": ["Monday"] }]
Pas	La météo est tout sauf « Raining »	"Weather": [{ "anything-but": ["Raining"] }]
Numérique (égal)	Le prix est de 100	"Price": [{ "numeric": ["=", 100] }]
Numérique (plage)	Le prix est supérieur à 10 et inférieur ou égal à 20	"Price": [{ "numeric": [">", 10, "<=", 20] }]
Existe	ProductName existe	ProductName« : [{« existe » : vrai}]
N'existe pas	ProductName n'existe pas	ProductName« : [{« existe » : faux}]
Commence par	La région se trouve aux États-Unis	"Region": [{"prefix": "us-" }]
Termine par	FileName se termine par une extension .png.	FileName« : [{« suffixe » : « .png »}]

> **Note**
>
> Comme pour les chaînes EventBridge, Lambda utilise une character-by-character correspondance exacte sans pliage en majuscules ni aucune autre normalisation des chaînes. Pour les nombres, Lambda utilise également une représentation par chaîne. Par exemple, 300, 300,0 et 3.0e2 ne sont pas considérés égaux.

Notez que l'opérateur Exists ne fonctionne que sur les nœuds feuilles de votre source d'événements JSON. Il ne tient pas compte des nœuds intermédiaires. Par exemple, avec le JSON suivant, le

modèle de filtre `{ "person": { "address": [ { "exists": true } ] } }` ne trouverait aucune correspondance, car `"address"` est un nœud intermédiaire.

```
{
  "person": {
    "name": "John Doe",
    "age": 30,
    "address": {
      "street": "123 Main St",
      "city": "Anytown",
      "country": "USA"
    }
  }
}
```

Attacher des critères de filtre à un mappage de sources d'événements (console)

Suivez ces étapes pour créer un nouveau mappage de source d'événement avec des critères de filtre à l'aide de la console Lambda.

Pour créer un nouveau mappage de sources d'événements avec des critères de filtre (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom d'une fonction pour laquelle créer un mappage de source d'événements.
3. Sous Function overview (Vue d'ensemble de la fonction), choisissez Add trigger (Ajouter un déclencheur).
4. Pour Trigger configuration (Configuration du déclencheur), choisissez un type de déclencheur qui prend en charge le filtrage des événements. Pour obtenir la liste des services pris en charge, reportez-vous à la liste figurant au début de cette page.
5. Développer Additional settings (Paramètres supplémentaires).
6. Sous Filter criteria (Critères de filtre), choisissez Add (Ajouter) puis définissez et saisissez vos filtres. Par exemple, vous pouvez saisir ce qui suit.

   ```
   { "Metadata" : [ 1, 2 ] }
   ```

Cela indique à Lambda de traiter uniquement les enregistrements où le champ `Metadata` est égal à 1 ou 2. Vous pouvez continuer à sélectionner Ajouter pour ajouter d'autres filtres jusqu'à la quantité maximale autorisée.

7. Lorsque vous avez terminé d'ajouter vos filtres, sélectionnez Enregistrer.

Lorsque vous saisissez des critères de filtrage à l'aide de la console, vous n'entrez que le modèle de filtrage et n'avez pas besoin d'indiquer la touche `Pattern` ni d'échapper aux guillemets. À l'étape 6 des instructions précédentes, `{ "Metadata" : [ 1, 2 ] }` correspond à ce qui suit `FilterCriteria`.

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

Après avoir créé le mappage de la source d'événements dans la console, vous pouvez voir les `FilterCriteria` formatés dans les détails du déclencheur. Pour plus d'exemples de création de filtres d'événements à l'aide de la console, consultez Utilisation de filtres avec différents Services AWS.

Attacher des critères de filtre à un mappage de sources d'événements (AWS CLI)

Supposons que vous souhaitiez que le mappage d'une source d'événements comporte les suivants `FilterCriteria`:

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

Pour créer un nouveau mappage de source d'événements avec ces critères de filtre à l'aide de AWS Command Line Interface (AWS CLI), exécutez la commande suivante.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"Metadata\" : [ 1, 2 ]}"}]}'
```

Cette create-event-source-mappingcommande crée un nouveau mappage de source d'événements Amazon SQS pour la fonction `my-function` avec la valeur spécifiée. `FilterCriteria`

Pour ajouter ces critères de filtre à un mappage des sources d'événements existant, exécutez la commande suivante.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"Metadata\" : [ 1, 2 ]}"}]}'
```

Notez que pour mettre à jour un mappage de source d'événements, vous avez besoin de son UUID. Vous pouvez obtenir l'UUID lors d'un list-event-source-mappingsappel. Lambda renvoie également l'UUID dans la réponse de la CLI. create-event-source-mapping

Pour supprimer les critères de filtre d'une source d'événement, vous pouvez exécuter la update-event-source-mappingcommande suivante avec un `FilterCriteria` objet vide.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria "{}"
```

Pour plus d'exemples de création de filtres d'événements à l'aide du AWS CLI, voirUtilisation de filtres avec différents Services AWS.

Attacher des critères de filtre à un mappage de sources d'événements (AWS SAM)

Supposons que vous souhaitiez configurer une source d'événements AWS SAM pour utiliser les critères de filtre suivants :

```
{
   "Filters": [
```

```
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

Pour ajouter ces critères de filtre à votre mappage des sources d'événements, insérez l'extrait suivant dans le modèle YAML de votre source d'événements.

```
FilterCriteria:
  Filters:
    - Pattern: '{"Metadata": [1, 2]}'
```

Pour plus d'informations sur la création et la configuration d'un AWS SAM modèle pour un mappage de sources d'événements, consultez la EventSourcesection du guide du AWS SAM développeur. Pour plus d'exemples de création de filtres d'événements à l'aide AWS SAM de modèles, consultezUtilisation de filtres avec différents Services AWS.

Chiffrement des critères de filtre

Par défaut, Lambda ne chiffre pas votre objet de critères de filtre. Dans les cas d'utilisation où vous pouvez inclure des informations sensibles dans votre objet de critères de filtre, vous pouvez utiliser votre propre clé KMS pour les chiffrer.

Une fois que vous avez chiffré votre objet de critères de filtre, vous pouvez afficher sa version en texte brut à l'aide d'un appel d'GetEventSourceMappingAPI. Vous devez disposer des autorisations `kms:Decrypt` nécessaires pour afficher correctement les critères de filtre en texte brut.

> **Note**
>
> Si votre objet de critères de filtre est chiffré, Lambda expédie la valeur du `FilterCriteria` champ dans la réponse aux appels. ListEventSourceMappings Ce champ s'affiche plutôt sous la forme `null`. Pour connaître la vraie valeur de`FilterCriteria`, utilisez l'GetEventSourceMappingAPI.
> Pour afficher la valeur déchiffrée de `FilterCriteria` dans la console, assurez-vous que votre rôle IAM contient des autorisations pour. GetEventSourceMapping

Vous pouvez spécifier votre propre clé KMS via la console, l'API/CLI ou CloudFormation.

Pour chiffrer les critères de filtre à l'aide d'une clé KMS appartenant au client (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez Add trigger (Ajouter déclencheur). Si vous possédez déjà un déclencheur, choisissez l'onglet Configuration, puis sur Déclencheurs. Sélectionnez le déclencheur existant, puis choisissez Modifier.
3. Cochez la case en regard de Chiffrer avec une clé KMS gérée par le client.
4. Pour Choisir une clé de chiffrement KMS gérée par le client, sélectionnez une clé activée existante ou créez-en une. Selon l'opération, vous avez besoin de certaines ou de toutes les autorisations suivantes : `kms:DescribeKey`, `kms:GenerateDataKey` et `kms:Decrypt`. Utilisez la stratégie de clé KMS pour octroyer ces autorisations.

Si vous utilisez votre propre clé KMS, les opérations d'API suivantes doivent être autorisées dans la stratégie de clé :

- `kms:Decrypt` : doit être octroyé au principal du service Lambda régional (`lambda.`*`AWS_region`*`.amazonaws.com`). Cela permet à Lambda de déchiffrer les données avec cette clé KMS.
  - Pour éviter un problème des adjoints confus entre services, la stratégie de clé utilise la clé de condition globale `aws:SourceArn`. La valeur correcte de la clé `aws:SourceArn` est l'ARN de votre ressource de mappage des sources d'événements. Vous ne pouvez donc l'ajouter à votre stratégie qu'une fois que vous connaissez son ARN. Lambda transmet également les clés `aws:lambda:FunctionArn` et `aws:lambda:EventSourceArn`, ainsi que leurs valeurs respectives dans le contexte de chiffrement lors d'une requête de déchiffrement à KMS. Ces valeurs doivent correspondre aux conditions spécifiées dans la stratégie de clé pour que la requête de déchiffrement aboutisse. Vous n'avez pas besoin d'inclure EventSourceArn les sources d'événements Kafka autogérées car elles n'ont pas de. EventSourceArn
- `kms:Decrypt`— Doit également être accordé au principal qui a l'intention d'utiliser la clé pour afficher les critères de filtrage en texte brut dans les appels GetEventSourceMappingd'DeleteEventSourceMappingAPI.
- `kms:DescribeKey` : fournit les détails des clés gérées par le client pour permettre au principal spécifié d'utiliser la clé.
- `kms:GenerateDataKey` :permet à Lambda de générer une clé de données pour chiffrer les critères de filtre, au nom du principal spécifié (chiffrement d'enveloppe).

Vous pouvez l'utiliser AWS CloudTrail pour suivre les AWS KMS demandes que Lambda effectue en votre nom. Pour des exemples CloudTrail d'événements, voir???.

Nous recommandons également d'utiliser la clé de condition `kms:ViaService` pour limiter l'utilisation de la clé KMS aux requêtes ne provenant que de Lambda. La valeur de cette clé est le principal de service Lambda régional (`lambda.`*`AWS_region`*`.amazonaws.com`). Voici un exemple de stratégie de clé qui accorde toutes les autorisations pertinentes :

Example AWS KMS politique clé

JSON

```
{
    "Version":"2012-10-17",
    "Id": "example-key-policy-1",
    "Statement": [
        {
            "Sid": "Allow Lambda to decrypt using the key",
            "Effect": "Allow",
            "Principal": {
                "Service": "lambda.us-east-1.amazonaws.com"
            },
            "Action": [
                "kms:Decrypt"
            ],
            "Resource": "*",
            "Condition": {
                "ArnEquals" : {
                    "aws:SourceArn": [
                        "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:<esm_uuid>"
                    ]
                },
                "StringEquals": {

 "kms:EncryptionContext:aws:lambda:FunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:test-function",

 "kms:EncryptionContext:aws:lambda:EventSourceArn": "arn:aws:sqs:us-
east-1:123456789012:test-queue"
                }
            }
```

```
        },
        {
            "Sid": "Allow actions by an AWS account on the key",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::123456789012:root"
            },
            "Action": "kms:*",
            "Resource": "*"
        },
        {
            "Sid": "Allow use of the key to specific roles",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::123456789012:role/ExampleRole"
            },
            "Action": [
                "kms:Decrypt",
                "kms:DescribeKey",
                "kms:GenerateDataKey"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals" : {
                    "kms:ViaService": "lambda.us-east-1.amazonaws.com"
                }
            }
        }
    ]
}
```

Pour utiliser votre propre clé KMS afin de chiffrer les critères de filtre, vous pouvez également utiliser la CreateEventSourceMapping AWS CLI commande suivante. Spécifiez l'ARN de la clé KMS avec l'option `--kms-key-arn`.

```
aws lambda create-event-source-mapping --function-name my-function \
    --maximum-batching-window-in-seconds 60 \
    --event-source-arn arn:aws:sqs:us-east-1:123456789012:my-queue \
    --filter-criteria "{\"filters\": [{\"pattern\": \"{\"a\": [\"1\", \"2\"]}\" }]}" \
    --kms-key-arn arn:aws:kms:us-east-1:123456789012:key/055efbb4-xmpl-4336-
ba9c-538c7d31f599
```

Si vous disposez déjà d'un mappage de source d'événements, utilisez plutôt la UpdateEventSourceMapping AWS CLI commande. Spécifiez l'ARN de la clé KMS avec l'option `--kms-key-arn`.

```
aws lambda update-event-source-mapping --function-name my-function \
    --maximum-batching-window-in-seconds 60 \
    --event-source-arn arn:aws:sqs:us-east-1:123456789012:my-queue \
    --filter-criteria "{\"filters\": [{\"pattern\": \"{\"a\": [\"1\", \"2\"]}\" }]}" \
    --kms-key-arn arn:aws:kms:us-east-1:123456789012:key/055efbb4-xmpl-4336-ba9c-538c7d31f599
```

Cette opération remplace toute clé KMS précédemment spécifiée. Si vous spécifiez l'option `--kms-key-arn` avec un argument vide, Lambda cesse d'utiliser votre clé KMS pour chiffrer les critères de filtre. Au lieu de cela, Lambda revient par défaut à l'utilisation d'une clé appartenant à Amazon.

Pour spécifier votre propre clé KMS dans un CloudFormation modèle, utilisez la `KMSKeyArn` propriété du type de `AWS::Lambda::EventSourceMapping` ressource. Par exemple, vous pouvez insérer l'extrait de code suivant dans le modèle YAML de votre source d'événement.

```
MyEventSourceMapping:
  Type: AWS::Lambda::EventSourceMapping
  Properties:
    ...
    FilterCriteria:
      Filters:
        - Pattern: '{"a": [1, 2]}'
    KMSKeyArn: "arn:aws:kms:us-east-1:123456789012:key/055efbb4-xmpl-4336-ba9c-538c7d31f599"
    ...
```

Pour pouvoir afficher vos critères de filtre chiffrés en texte clair dans un appel d'DeleteEventSourceMappingAPI GetEventSourceMappingou un appel d'API, vous devez disposer d'`kms:Decrypt`autorisations.

À compter du 6 août 2024, le `FilterCriteria` champ n'apparaît plus dans les AWS CloudTrail journaux provenant de CreateEventSourceMappinget dans UpdateEventSourceMappingles appels d'DeleteEventSourceMappingAPI si votre fonction n'utilise pas le filtrage des événements. Si votre fonction utilise le filtrage des événements, le champ `FilterCriteria` apparaît vide (`{}`). Vous pouvez toujours afficher vos critères de filtre en texte clair en réponse aux appels d'GetEventSourceMappingAPI si vous êtes autorisé à `kms:Decrypt` utiliser la bonne clé KMS.

Exemple d'entrée de CloudTrail journal pour les Create/Update/DeleteEventSourceMapping appels

Dans l' AWS CloudTrail exemple d'entrée de journal suivant pour un CreateEventSourceMapping appel, `FilterCriteria` apparaît comme vide (`{}`) car la fonction utilise le filtrage des événements. C'est le cas même si l'objet `FilterCriteria` contient des critères de filtre valides que votre fonction utilise activement. Si la fonction n'utilise pas le filtrage des événements, le `FilterCriteria` champ CloudTrail n'apparaîtra pas du tout dans les entrées du journal.


```
{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROA123456789EXAMPLE:userid1",
        "arn": "arn:aws:sts::123456789012:assumed-role/Example/example-role",
        "accountId": "123456789012",
        "accessKeyId": "ASIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROA987654321EXAMPLE",
                "arn": "arn:aws:iam::123456789012:role/User1",
                "accountId": "123456789012",
                "userName": "User1"
            },
            "webIdFederationData": {},
            "attributes": {
                "creationDate": "2024-05-09T20:35:01Z",
                "mfaAuthenticated": "false"
            }
        },
        "invokedBy": "AWS Internal"
    },
    "eventTime": "2024-05-09T21:05:41Z",
    "eventSource": "lambda.amazonaws.com",
    "eventName": "CreateEventSourceMapping20150331",
    "awsRegion": "us-east-2",
    "sourceIPAddress": "AWS Internal",
    "userAgent": "AWS Internal",
    "requestParameters": {
        "eventSourceArn": "arn:aws:sqs:us-east-2:123456789012:example-queue",
        "functionName": "example-function",
        "enabled": true,
        "batchSize": 10,
```

```
        "filterCriteria": {},
        "kMSKeyArn": "arn:aws:kms:us-east-2:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111",
        "scalingConfig": {},
        "maximumBatchingWindowInSeconds": 0,
        "sourceAccessConfigurations": []
    },
    "responseElements": {
        "uUID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
        "batchSize": 10,
        "maximumBatchingWindowInSeconds": 0,
        "eventSourceArn": "arn:aws:sqs:us-east-2:123456789012:example-queue",
        "filterCriteria": {},
        "kMSKeyArn": "arn:aws:kms:us-east-2:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:example-
function",
        "lastModified": "May 9, 2024, 9:05:41 PM",
        "state": "Creating",
        "stateTransitionReason": "USER_INITIATED",
        "functionResponseTypes": [],
        "eventSourceMappingArn": "arn:aws:lambda:us-east-2:123456789012:event-source-
mapping:a1b2c3d4-5678-90ab-cdef-EXAMPLEbbbbb"
    },
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLE33333",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLE22222",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "123456789012",
    "eventCategory": "Management",
    "sessionCredentialFromConsole": "true"
}
```


Utilisation de filtres avec différents Services AWS

Les différents types de sources d'événements utilisent des valeurs clés différentes pour leurs champs de données. Pour filtrer sur les propriétés de données, assurez-vous d'utiliser la bonne clé dans le modèle de votre filtre. Le tableau suivant indique les clés de filtrage pour chacune des clés prises en charge Service AWS.

Service AWS	Clé de filtrage
DynamoDB	`dynamodb`
Kinesis	`data`
Amazon MQ	`data`
Amazon MSK	`value`
Self-managed Apache Kafka	`value`
Amazon SQS	`body`

Les sections suivantes donnent des exemples de modèles de filtre pour différents types de sources d'événements. Elles fournissent également des définitions des formats de données entrantes pris en charge et des formats de corps de modèle de filtre pour chaque service pris en charge.

- the section called “Filtrage des événements”
- the section called “Filtrage des événements”
- the section called “Filtrage des événements”
- the section called “Filtrage des événements”
- the section called “Filtrage des événements”

Test des fonctions Lambda dans la console

Vous pouvez tester votre fonction Lambda dans la console en invoquant votre fonction avec un événement de test. Un événement de test est une entrée JSON pour votre fonction. Si votre fonction ne nécessite pas d'entrée, l'événement peut être un document vide (`{}`).

Lorsque vous exécutez un test dans la console, Lambda invoque votre fonction de manière synchrone avec l'événement de test. L'exécution de la fonction convertit le JSON de l'événement en un objet et le transmet à la méthode du gestionnaire de votre code afin qu'il soit traité.

> **Créer un événement de test**
>
> Avant de pouvoir tester dans la console, vous devez créer un événement de test privé ou partageable.

Invocation de fonctions avec des événements de test

Pour tester une fonction

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom de la fonction que vous voulez tester.
3. Choisissez l'onglet Test.
4. Sous Test event (Événement de test), sélectionnez Create new event (Créer un nouvel événement) ou Edit saved event (Modifier un événement sauvegardé), puis sélectionnez l'événement sauvegardé que vous voulez utiliser.
5. En option, choisissez un modèle pour le JSON de l'événement.
6. Sélectionnez Tester).
7. Pour examiner les résultats du test, sous Execution result (Résultat de l'exécution), développez Details (Détails).

Pour invoquer votre fonction sans enregistrer votre événement de test, choisissez Tester avant d'enregistrer. Cela crée un événement de test non sauvegardé que Lambda ne conserve que pour la durée de la session.

Pour les environnements d'exécution Node.js, Python et Ruby, vous pouvez également accéder à vos événements de test enregistrés et non enregistrés dans l'onglet Code. Utilisez la section ÉVÉNEMENTS DE TEST pour créer, modifier et exécuter des tests.

Création d'événements de test privés

Les événements de test privés sont disponibles uniquement pour le créateur de l'événement et ils ne nécessitent aucune autorisation supplémentaire pour être utilisés. Vous pouvez créer et enregistrer jusqu'à 10 événements de test privés par fonction.

Pour créer un événement de test privé

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom de la fonction que vous voulez tester.
3. Choisissez l'onglet Test.
4. Sous Test event (Événement de test), procédez comme suit :
    a. Choisissez un Template (Modèle).
    b. Saisissez un Name (Nom) pour le test.
    c. Dans la zone de saisie de texte, entrez l'événement de test JSON.
    d. Sous Event sharing settings (Paramètres de partage de l'événement), choisissez Private (Privé).
5. Sélectionnez Enregistrer les modifications.

Pour les environnements d'exécution Node.js, Python et Ruby, vous pouvez également créer des événements de test dans l'onglet Code. Utilisez la section ÉVÉNEMENTS DE TEST pour créer, modifier et exécuter des tests.

Création d'événements de test partageables

Les événements de test partageables sont des événements de test que vous pouvez partager avec d'autres utilisateurs du même AWS compte. Vous pouvez modifier les événements de test partageables des autres utilisateurs et invoquer votre fonction avec eux.

Lambda enregistre les événements de test partageables sous forme de schémas dans un registre de schémas Amazon EventBridge (CloudWatch Events) nommé. `lambda-testevent-schemas` Comme Lambda utilise ce registre pour stocker et appeler les événements de test partageables que

vous créez, nous vous recommandons de ne pas modifier ce registre ou de créer un registre utilisant le nom `lambda-testevent-schemas`.

Pour consulter, partager et modifier des événements de test partageables, vous devez disposer d'autorisations pour toutes les opérations d'API de registre de schéma EventBridge (CloudWatch événements) suivantes :

- `schemas.CreateRegistry`
- `schemas.CreateSchema`
- `schemas.DeleteSchema`
- `schemas.DeleteSchemaVersion`
- `schemas.DescribeRegistry`
- `schemas.DescribeSchema`
- `schemas.GetDiscoveredSchema`
- `schemas.ListSchemaVersions`
- `schemas.UpdateSchema`

Notez que l'enregistrement des modifications apportées à un événement de test partageable remplace cet événement.

Si vous ne pouvez pas créer, modifier ou consulter des événements de test partageables, vérifiez que votre compte dispose des autorisations requises pour ces opérations. Si vous disposez des autorisations requises mais que vous ne pouvez toujours pas accéder aux événements de test partageables, vérifiez s'il existe des politiques basées sur les ressources susceptibles de limiter l'accès au registre EventBridge (des CloudWatch événements).

Pour créer un événement de test partageable

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom de la fonction que vous voulez tester.
3. Choisissez l'onglet Test.
4. Sous Test event (Événement de test), procédez comme suit :
    a. Choisissez un Template (Modèle).
    b. Saisissez un Name (Nom) pour le test.
    c. Dans la zone de saisie de texte, entrez l'événement de test JSON.

d. Sous Event sharing settings (Paramètres de partage de l'événement), choisissez Shareable (Partageable).

5. Sélectionnez Enregistrer les modifications.

> Utilisez des événements de test partageables avec AWS Serverless Application Model.
>
> Vous pouvez l'utiliser AWS SAM pour invoquer des événements de test partageables. Pour en savoir plus, consultez `sam remote test-event` le Guide du développeur de AWS Serverless Application Model

Suppression des schémas d'événements de test partageables

Lorsque vous supprimez des événements de test partageables, Lambda les supprime du registre `lambda-testevent-schemas`. Si vous supprimez le dernier événement de test partageable du registre, Lambda supprime le registre.

Si vous supprimez la fonction, Lambda ne supprime pas les schémas d'événements de test partageables associés. Vous devez nettoyer ces ressources manuellement depuis la console EventBridge (CloudWatch Events).

États de la fonction Lambda

Lambda inclut un champ État dans la configuration de la fonction pour toutes les fonctions, afin d'indiquer quand la fonction est prête à être invoquée. `State` fournit des informations sur l'état actuel de la fonction, y compris si vous pouvez invoquer la fonction avec succès. Les états de la fonction ne modifient pas le comportement des appels de la fonction ou la façon dont votre fonction exécute le code.

> **Note**
>
> Les définitions de l'état des fonctions diffèrent légèrement selon les SnapStartfonctions. Pour de plus amples informations, veuillez consulter Lambda SnapStart et états des fonctions.

Dans de nombreux cas, une table DynamoDB constitue un moyen idéal de conservation d'état entre invocations, car elle fournit un accès aux données à faible latence et peut se mettre à l'échelle avec le service Lambda. Vous pouvez également stocker des données dans Amazon EFS pour Lambda si vous utilisez ce service, ce qui permet un accès à faible latence au stockage du système de fichiers.

Les états de la fonction incluent :

- `Pending` : une fois que Lambda crée la fonction, il définit l'état comme « en attente ». Lorsqu'elle est en attente, Lambda tente de créer ou de configurer des ressources pour la fonction, telles que des ressources VPC ou EFS. Lambda n'appelle pas de fonction pendant que l'état est en attente. Les appels ou autres actions d'API qui agissent sur la fonction échoueront.
- `Active` – La fonction passera à l'état actif lorsque Lambda aura terminé la configuration et l'approvisionnement des ressources. Les fonctions peuvent uniquement être appelées avec succès lorsqu'elles sont actives.
- `Failed` : indique que la configuration ou l'approvisionnement des ressources ont rencontré une erreur. Lorsque la création d'une fonction échoue, Lambda définit l'état de la fonction sur échec, et vous devez supprimer et recréer la fonction.
- `Inactive` : une fonction devient inactive lorsqu'elle a été inactive suffisamment longtemps afin que Lambda puisse récupérer les ressources externes qui ont été configurées pour elle. Lorsque vous essayez d'appeler une fonction qui est inactive, l'invocation échoue et Lambda définit la fonction à l'état « en attente » jusqu'à ce que les ressources de la fonction soient recréées. Si Lambda ne parvient pas à recréer les ressources, la fonction retourne à l'état inactif. Il se peut que vous deviez résoudre les erreurs et redéployer votre fonction pour la rétablir à l'état actif.

Si vous utilisez des flux de travail automatisés basés sur le SDK ou si vous appelez APIs directement le service Lambda, assurez-vous de vérifier l'état d'une fonction avant de l'invoquer pour vérifier qu'elle est active. Vous pouvez le faire à l'aide de l'action GetFunctionde l'API Lambda ou en configurant un serveur à l'aide du SDK pour AWS Java 2.0.

```
aws lambda get-function --function-name my-function --query 'Configuration.[State,
 LastUpdateStatus]'
```

Vous devriez voir la sortie suivante :

```
[
 "Active",
 "Successful"
]
```

Les opérations suivantes échouent lorsque la création de fonction est en attente :

- Invoke
- UpdateFunctionCode
- UpdateFunctionConfiguration
- PublishVersion

États de fonction lors des mises à jour

Lambda effectue deux opérations pour la mise à jour des fonctions :

- UpdateFunctionCode: met à jour le package de déploiement de la fonction
- UpdateFunctionConfiguration: met à jour la configuration de la fonction

Lambda utilise LastUpdateStatuscet attribut pour suivre la progression de ces opérations de mise à jour. Quand une mise à jour est en cours (`"LastUpdateStatus": "InProgress"`) :

- L'État de la fonction reste `Active`.
- Les invocations continuent d'utiliser le code et la configuration précédents de la fonction jusqu'à ce que la mise à jour soit terminée.
- Les opérations suivantes échouent :

- UpdateFunctionCode
- UpdateFunctionConfiguration
- PublishVersion
- TagResource

Lorsqu'une mise à jour échoue (quand`"LastUpdateStatus": "Failed"`) :

- L'État de la fonction reste `Active`.
- Les invocations continuent d'utiliser le code et la configuration précédents de la fonction.

Example GetFunctionConfiguration réponse

L'exemple suivant est le résultat d'une GetFunctionConfigurationdemande sur une fonction en cours de mise à jour.

```
{
    "FunctionName": "my-function",
    "FunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function",
    "Runtime": "nodejs24.x",
    "VpcConfig": {
        "SubnetIds": [
            "subnet-071f712345678e7c8",
            "subnet-07fd123456788a036",
            "subnet-0804f77612345cacf"
        ],
        "SecurityGroupIds": [
            "sg-085912345678492fb"
        ],
        "VpcId": "vpc-08e1234569e011e83"
    },
    "State": "Active",
    "LastUpdateStatus": "InProgress",
    ...
}
```

Présentation du comportement des nouvelles tentatives dans Lambda

Lorsque vous invoquez une fonction directement, vous déterminez la stratégie de gestion des erreurs liées au code de votre fonction. Lambda ne réessaie pas automatiquement ce type d'erreur en votre nom. Pour réessayer, vous pouvez manuellement invoquer à nouveau votre fonction, envoyer l'événement échoué à une file d'attente pour le débogage, ou ignorer l'erreur. Le code de votre fonction peut s'exécuter entièrement, partiellement ou pas du tout. Si vous recommencez, assurez-vous que le code de votre fonction peut gérer le même événement plusieurs fois, sans provoquer des transactions en double ou d'autres effets secondaires.

Lorsque vous appelez une fonction indirectement, vous devez connaître le comportement de la nouvelle tentative de l'appelant et tout service que la demande rencontre sur le chemin. Cela inclut les scénarios suivants.

- Appel asynchrone – Lambda réessaie deux fois les erreurs de fonction. Si la fonction ne dispose pas de la capacité suffisante pour gérer toutes les requêtes entrantes, des événements peuvent attendre dans la file d'attente pendant des heures ou des jours avant d'être envoyés à la fonction. Vous pouvez configurer une file d'attente de lettres mortes sur la fonction pour capturer les événements qui n'ont pas été traitées avec succès. Pour de plus amples informations, consultez the section called "Files d'attente de lettres mortes".
- Mappages de source d'événement – Les mappages de source d'événement qui lisent à partir de flux réessaient la totalité du lot d'événements. Les erreurs répétées bloquent le traitement de la partition concernée jusqu'à la résolution de l'erreur ou l'expiration des éléments. Pour détecter des partitions bloquées, vous pouvez surveiller la métrique Âge de l'itérateur.

  Pour les mappages de source d'événement qui lisent à partir d'une file d'attente, vous devez déterminer l'intervalle entre les nouvelles tentatives et la destination pour les événements ayant échoué, en configurant le délai de visibilité et la stratégie de redirection dans la file d'attente source. Pour plus d'informations, consultez Procédure de traitement par Lambda des enregistrements provenant de sources d'événements basées sur des flux et des files d'attente ainsi que les rubriques spécifiques du service dans Invoquer Lambda avec des événements provenant d'autres services AWS.
- Services AWS – Des services AWS peuvent appeler votre fonction de manière synchrone ou asynchrone. Dans le cas des appels synchrones, le service décide s'il convient d'effectuer une nouvelle tentative. Par exemple, les opérations par lot Amazon S3 retentent l'opération si la fonction Lambda renvoie un code de réponse `TemporaryFailure`. Les services avec requêtes

proxy d’un utilisateur ou client en amont peuvent également avoir une stratégie de nouvelle tentative ou relayer la réponse d’erreur au demandeur. Par exemple, API Gateway transmet toujours la réponse d’erreur au demandeur.

Pour une invocation asynchrone, la logique de nouvelle tentative est la même, quelle que soit la source de l’invocation. Par défaut, Lambda réessaie une invocation asynchrone qui a échoué jusqu’à deux fois. Pour de plus amples informations, consultez Méthode de gestion des erreurs et nouvelles tentatives d’invocation asynchrone par Lambda.

- Autres comptes et clients – Lorsque vous accordez l'accès à d'autres comptes, vous pouvez utiliser des stratégies basées sur une ressource afin de restreindre les services ou ressources qu'ils peuvent configurer pour appeler votre fonction. Pour protéger votre fonction de toute surcharge, envisagez de placer une couche d'API devant votre fonction avec Amazon API Gateway.

Pour vous aider à gérer les erreurs dans les applications Lambda, Lambda s'intègre avec des services tels que Amazon CloudWatch et AWS X-Ray. Vous pouvez utiliser une combinaison de journaux, de mesures, d'alarmes et de suivi, pour détecter rapidement et d'identifier les problèmes dans le code de votre fonction, de l'API ou d'autres ressources qui prennent en charge votre application. Pour de plus amples informations, consultez Surveillance, débogage et dépannage des fonctions Lambda.

Utilisation de la détection de boucle récursive Lambda pour prévenir les boucles infinies

Lorsque vous configurez une fonction Lambda pour qu'elle envoie une sortie au même service ou à la même ressource que celui ou celle qui l'invoque, il est possible de créer une boucle récursive infinie. Par exemple, une fonction Lambda peut écrire un message dans une file d'attente Amazon Simple Queue Service (Amazon SQS), qui invoque ensuite la même fonction. Cette invocation amène la fonction à écrire un autre message dans la file d'attente, qui à son tour invoque à nouveau la fonction.

Les boucles récursives involontaires peuvent entraîner la facturation de frais imprévus à votre charge. Compte AWS Les boucles peuvent également entraîner la mise à l'échelle de Lambda et l'utilisation de toute la simultanéité disponible de votre compte. Pour réduire l'impact des boucles non intentionnelles, Lambda détecte certains types de boucles récursives peu de temps après leur apparition. Par défaut, lorsque Lambda détecte une boucle récursive, il arrête l'invocation de votre fonction et vous en informe. Si votre conception utilise intentionnellement des schémas récursifs, vous pouvez modifier la configuration par défaut d'une fonction pour permettre à celle-ci d'être invoquée de manière récursive. Pour plus d'informations, consultez the section called "Octroi d'exécution d'une fonction Lambda dans une boucle récursive".

Sections

- Comprendre la détection de boucles récursives
- Soutenu Services AWS et SDKs
- Notifications de boucles récursives
- Répondre aux notifications de détection de boucles récursives
- Octroi d'exécution d'une fonction Lambda dans une boucle récursive
- Régions prises en charge pour la détection des boucles récursives de Lambda

Comprendre la détection de boucles récursives

La détection de boucles récursives dans Lambda fonctionne en suivant les événements. Lambda est un service de calcul piloté par les événements qui exécute le code de votre fonction lorsque certains événements se produisent. Par exemple, lorsqu'un élément est ajouté à une file d'attente Amazon SQS ou à une rubrique Amazon Simple Notification Service (Amazon SNS). Lambda transmet les événements à votre fonction sous forme d'objets JSON, qui contiennent des informations sur la

modification d'état du système. Lorsqu'un événement provoque l'exécution de votre fonction, cela s'appelle une invocation.

Pour détecter les boucles récursives, Lambda utilise des en-têtes de suivi AWS X-Ray. Lorsque les Services AWS qui prennent en charge cette la détection des boucles récursives envoient des événements à Lambda, ces événements sont automatiquement annotés avec des métadonnées. Lorsque votre fonction Lambda écrit l'un de ces événements dans un autre événement pris en charge à Service AWS l'aide d'une version prise en charge d'un AWS SDK, elle met à jour ces métadonnées. Les métadonnées mises à jour comprennent un décompte du nombre de fois où l'événement a invoqué la fonction.

> **Note**
>
> Il n'est pas nécessaire d'activer le suivi actif de X-Ray pour que cette fonctionnalité soit active. La détection des boucles récursives est activée par défaut pour tous les clients AWS . L'utilisation de cette fonctionnalité est gratuite.

Une chaîne de requêtes est une séquence d'invocations Lambda provoquée par le même événement déclencheur. Par exemple, imaginez qu'une file d'attente Amazon SQS invoque votre fonction Lambda. Votre fonction Lambda renvoie ensuite l'événement traité à la même file d'attente Amazon SQS, qui invoque à nouveau votre fonction. Dans cet exemple, chaque invocation de votre fonction appartient à la même chaîne de requêtes.

Si votre fonction est invoquée environ 16 fois dans la même chaîne de requêtes, Lambda arrête automatiquement l’invocation de fonction suivante dans cette chaîne de requêtes et vous en informe. Si votre fonction est configurée avec plusieurs déclencheurs, les invocations provenant d'autres déclencheurs ne sont pas affectées.

> **Note**
>
> Même lorsque le paramètre `maxReceiveCount` de la politique de redirection de la file d’attente source est supérieur à 16, la protection contre la récursion Lambda n’empêche pas Amazon SQS de réessayer le message après la détection et l’arrêt d’une boucle récursive. Lorsque Lambda détecte une boucle récursive et interrompt les invocations suivantes, il renvoie une `RecursiveInvocationException` au mappage des sources d'événements. Cela augmente la valeur `receiveCount` du message. Lambda continue de réessayer le message et de bloquer les invocations de fonctions jusqu’à ce qu’Amazon SQS détermine

que le délai `maxReceiveCount` est dépassé et envoie le message à la file d'attente de lettres mortes configurée.

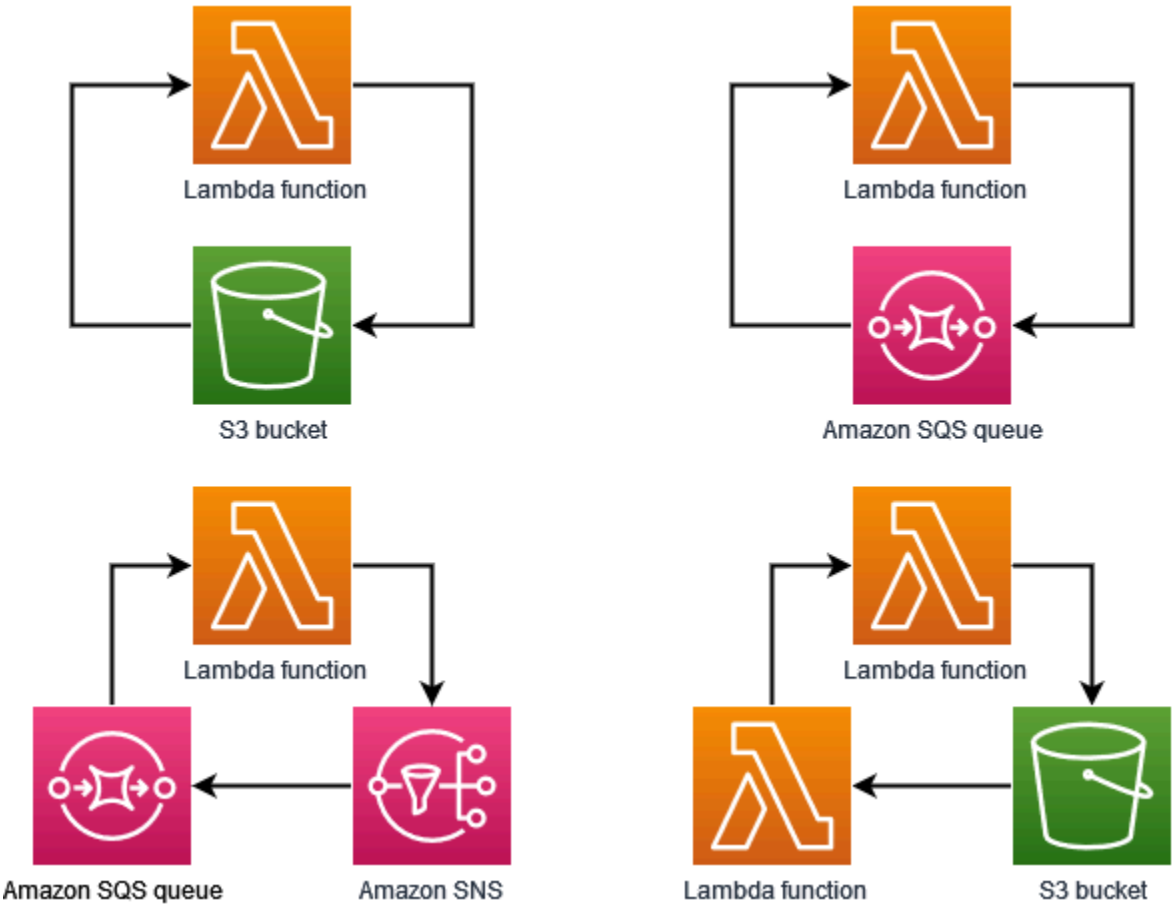
Si vous avez configuré une destination en cas d'échec ou une file d'attente de lettres mortes pour votre fonction, Lambda envoie également l'événement de l'invocation arrêtée vers votre destination ou votre file d'attente de lettres mortes. Lorsque vous configurez une file d'attente de destination ou une file d'attente de lettres mortes pour votre fonction, assurez-vous de ne pas utiliser un déclencheur d'événement ou mappage des sources d'événements que votre fonction utilise également. Si vous envoyez des événements à la même ressource qui invoque votre fonction, vous pouvez créer une autre boucle récursive, et cette boucle sera également résiliée. Si vous désactivez la détection des boucles récursives, cette boucle ne sera pas interrompue.

Soutenu Services AWS et SDKs

Lambda ne peut détecter que les boucles récursives dont certaines sont prises en charge. Services AWS Pour que les boucles récursives soient détectées, votre fonction doit également utiliser l'une des boucles prises en charge AWS SDKs.

Soutenu Services AWS

Lambda détecte actuellement les boucles récursives entre vos fonctions, Amazon SQS, Amazon S3 et Amazon SNS. Lambda détecte également les boucles composées uniquement de fonctions Lambda, qui peuvent s'invoquer de manière synchrone ou asynchrone. Les schémas suivants montrent quelques exemples de boucles que Lambda peut détecter :



Lorsqu'une autre Service AWS , telle qu'Amazon DynamoDB, fait partie de la boucle, Lambda ne peut actuellement pas la détecter ni l'arrêter.

Comme Lambda ne détecte actuellement que les boucles récursives impliquant Amazon SQS, Amazon S3 et Amazon SNS, il est toujours possible que des boucles impliquant d' Services AWS autres boucles entraînent une utilisation involontaire de vos fonctions Lambda.

Pour éviter que des frais imprévus ne vous soient facturés Compte AWS, nous vous recommandons de configurer les CloudWatchalarmes Amazon pour vous avertir des habitudes d'utilisation inhabituelles. Par exemple, vous pouvez configurer pour vous CloudWatch avertir des pics de simultanéité ou d'invocation des fonctions Lambda. Vous pouvez également configurer une alarme de facturation pour vous avertir lorsque les dépenses sur votre compte dépassent un seuil que vous avez spécifié. Vous pouvez également utiliser AWS Cost Anomaly Detectionpour vous alerter en cas de schémas de facturation inhabituels.

Soutenu AWS SDKs

Pour que Lambda détecte les boucles récursives, votre fonction doit utiliser l'une des versions suivantes ou plus récentes du kit SDK :

Environnement d’exécution	Version minimale du AWS SDK requise
Node.js	2.1147.0 (kit SDK version 2) 3.105.0 (kit SDK version 3)
Python	1.24.46 (boto3) 1.27.46 (botocore)
Java 8 et Java 11	2,17135
Java 17	2,20,81
Java 21	2,21,24
.NET	3,7,293,0
Ruby	3,134,0
PHP	3,232,0
Go	SDK V2 1.57.0

Certains environnements d'exécution Lambda tels que Python et Node.js incluent une version du SDK. AWS Si la version du kit SDK incluse dans l'exécution de votre fonction est inférieure au minimum requis, vous pouvez ajouter une version prise en charge du kit SDK au package de déploiement de votre fonction. Vous pouvez également ajouter une version du kit SDK compatible à votre fonction à l'aide d'une couche Lambda. Pour obtenir la liste des composants SDKs inclus dans chaque environnement d'exécution Lambda, consultez. Environnements d’exécution (runtimes) Lambda

Notifications de boucles récursives

Lorsque Lambda arrête une boucle récursive, vous recevez des notifications le Tableau de bord Health et par e-mail. Vous pouvez également utiliser CloudWatch des métriques pour surveiller le nombre d'appels récursifs interrompus par Lambda.

Tableau de bord Health notifications

Lorsque Lambda arrête un appel récursif, une notification s' Tableau de bord Health affiche sur la page de santé de votre compte, sous Problèmes ouverts et récents. Notez qu'il peut s'écouler jusqu'à trois heures et demie après l'arrêt d'une invocation récursive par Lambda avant que cette notification ne s'affiche. Pour plus d'informations sur l'affichage des événements du compte dans le Tableau de bord Health, consultez Getting started with your AWS Health Dashboard — Your account health dans le AWS Health User Guide.

Alertes par e-mail

Lorsque Lambda arrête pour la première fois une invocation récursive de votre fonction, il vous envoie une alerte par e-mail. Lambda envoie au maximum un e-mail toutes les 24 heures pour chaque fonction de votre Compte AWS. Une fois que Lambda a envoyé une notification par e-mail, vous ne recevrez plus d'e-mails pour cette fonction pendant 24 heures, même si Lambda arrête d'autres invocations récursives de la fonction. Notez qu'il peut s'écouler jusqu'à trois heures et demie après l'arrêt d'une invocation récursive par Lambda avant que vous ne receviez cette alerte par e-mail.

Lambda envoie des alertes par e-mail en boucle récursive au contact principal Compte AWS de votre compte et au contact opérationnel alternatif. Pour plus d'informations sur l'affichage ou la mise à jour des adresses e-mail de votre compte, consultez la section Mise à jour des informations de contact dans la Référence générale AWS .

CloudWatch Métriques Amazon

La CloudWatch métrique `RecursiveInvocationsDropped` enregistre le nombre d'invocations de fonctions que Lambda a arrêtées parce que votre fonction a été invoquée plus de 16 fois environ au cours d'une même chaîne de requêtes. Lambda émet cette métrique dès qu'il arrête une invocation récursive. Pour afficher cette métrique, suivez les instructions relatives à l'affichage des métriques sur la CloudWatch console et choisissez la métrique`RecursiveInvocationsDropped`.

Répondre aux notifications de détection de boucles récursives

Lorsque votre fonction est invoquée environ plus de 16 fois par le même événement déclencheur, Lambda arrête l'invocation de la prochaine fonction pour cet événement afin de mettre fin à la boucle récursive. Pour éviter qu'une boucle récursive interrompue par Lambda ne se reproduise, procédez comme suit :

- Réduisez la simultanéité disponible de votre fonction à zéro, ce qui limite toutes les invocations futures.
- Supprimez ou désactivez le déclencheur ou le mappage des sources d'événements qui invoque votre fonction.
- Identifiez et corrigez les défauts de code qui réécrivent les événements dans la AWS ressource qui appelle votre fonction. Une source fréquente de défauts survient lorsque vous utilisez des variables pour définir la source et la cible des événements d'une fonction. Vérifiez que vous n'utilisez pas la même valeur pour les deux variables.

En outre, si la source d'événements pour votre fonction Lambda est une file d'attente Amazon SQS, envisagez de configurer une file d'attente de lettres mortes dans la file d'attente source.

> **Note**
>
> Veillez à configurer la file d'attente de lettres mortes sur la file d'attente source, non sur la fonction Lambda. La file d'attente de lettres mortes que vous configurez dans une fonction est utilisée pour la file d'attente d'invocation asynchrone et non pour les files d'attente source d'événement.

Si la source de l'événement est une rubrique Amazon SNS, pensez à ajouter une destination en cas d'échec pour votre fonction.

Pour réduire à zéro la simultanéité disponible de votre fonction (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom de votre fonction .
3. Choisissez Limiter.
4. Dans la boîte de dialogue Limiter votre fonction, choisissez Confirmer.

Pour supprimer un déclencheur ou un mappage des sources d'événements pour votre fonction (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom de votre fonction .
3. Choisissez l'onglet Configuration, puis Déclencheurs.
4. Sous Déclencheurs, sélectionnez le déclencheur ou le mappage des sources d'événements que vous souhaitez supprimer, puis choisissez Supprimer.
5. Dans la boîte de dialogue Supprimer des déclencheurs, choisissez Supprimer.

Pour désactiver un mappage des sources d'événements pour votre fonction (AWS CLI)

1. Pour trouver l'UUID du mappage des sources d'événements que vous souhaitez désactiver, exécutez la commande AWS Command Line Interface (AWS CLI). list-event-source-mappings

   ```
   aws lambda list-event-source-mappings
   ```

2. Pour désactiver le mappage des sources d'événements, exécutez la AWS CLI update-event-source-mappingcommande suivante.

   ```
   aws lambda update-event-source-mapping --function-name MyFunction \
   --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 --no-enabled
   ```

Octroi d’exécution d’une fonction Lambda dans une boucle récursive

Si votre conception utilise intentionnellement une boucle récursive, vous pouvez configurer une fonction Lambda pour permettre son invocation récursive. Nous vous recommandons d’éviter d’utiliser des boucles récursives dans votre conception. Les erreurs de mise en œuvre peuvent entraîner des appels récursifs utilisant toutes les données disponibles simultanément et des frais imprévus facturés sur votre Compte AWS compte.

> **Important**
>
> Si vous utilisez des boucles récursives, traitez-les avec prudence. Mettez en œuvre des barrières de protection conformes aux pratiques exemplaires afin de minimiser les risques d’erreurs de mise en œuvre. Pour en savoir plus sur les pratiques exemplaires en matière

d'utilisation de modèles d'invocation récursifs, consultez Recursive patterns that cause run-away Lambda functions sur Serverless Land.

Vous pouvez configurer des fonctions pour autoriser les boucles récursives à l'aide de la console Lambda, AWS Command Line Interface du AWS CLI() et PutFunctionRecursionConfigde l'API. Vous pouvez également configurer le paramètre de détection de boucle récursive d'une fonction dans AWS SAM et CloudFormation.

Par défaut, Lambda détecte et arrête les boucles récursives. À moins que votre conception n'utilise intentionnellement une boucle récursive, nous vous recommandons de ne pas modifier la configuration par défaut de vos fonctions.

Notez que lorsque vous configurez une fonction pour autoriser les boucles récursives, la CloudWatch métrique `RecursiveInvocationsDropped` n'est pas émise.

Console

Pour autoriser l'exécution d'une fonction dans une boucle récursive (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom de votre fonction pour ouvrir la page détaillée de votre fonction.
3. Choisissez l'onglet Configuration, puis sélectionnez Concurrence et détection de récursivité.
4. En regard de Détection de boucle récursive, choisissez Modifier.
5. Sélectionnez Autoriser les boucles récursives.
6. Choisissez Enregistrer.

AWS CLI

Vous pouvez utiliser l'PutFunctionRecursionConfigAPI pour permettre à votre fonction d'être invoquée dans une boucle récursive. Spécifiez `Allow` pour le paramètre de boucle récursive. Par exemple, vous pouvez appeler cette API avec la `put-function-recursion-config` AWS CLI commande suivante :

```
aws lambda put-function-recursion-config --function-name yourFunctionName --
recursive-loop Allow
```

Vous pouvez rétablir la configuration par défaut de votre fonction afin que Lambda mette fin aux boucles récursives lorsqu'il les détecte. Modifiez la configuration de votre fonction à l'aide de la console Lambda ou de l' AWS CLI.

Console

Pour configurer une fonction de manière à ce que les boucles récursives soient interrompues (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom de votre fonction pour ouvrir la page détaillée de votre fonction.
3. Choisissez l'onglet Configuration, puis sélectionnez Concurrence et détection de récursivité.
4. En regard de Détection de boucle récursive, choisissez Modifier.
5. Sélectionnez Mettre fin aux boucles récursives.
6. Choisissez Enregistrer.

AWS CLI

Vous pouvez utiliser l'PutFunctionRecursionConfigAPI pour configurer votre fonction afin que Lambda mette fin aux boucles récursives lorsqu'il les détecte. Spécifiez `Terminate` pour le paramètre de boucle récursive. Par exemple, vous pouvez appeler cette API avec la `put-function-recursion-config` AWS CLI commande suivante :

```
aws lambda put-function-recursion-config --function-name yourFunctionName --
recursive-loop Terminate
```

Régions prises en charge pour la détection des boucles récursives de Lambda

La détection des boucles récursives Lambda est prise en charge dans toutes les régions commerciales, à l'exception des régions Mexique (Centre) et Asie-Pacifique (Nouvelle-Zélande).

Création et gestion de la fonction Lambda URLs

Une URL de fonction est un point de terminaison HTTP(S) dédié pour votre fonction Lambda. Vous pouvez créer et configurer une URL de fonction via la console Lambda ou l'API Lambda.

> **Tip**
>
> Lambda propose deux méthodes pour appeler votre fonction via un point de terminaison HTTP : function URLs et Amazon API Gateway. Si vous ne savez pas quelle est la meilleure méthode pour votre cas d'utilisation, consultez the section called " URLs Function et Amazon API Gateway".

Lorsque vous créez une URL de fonction, Lambda génère automatiquement un point de terminaison URL unique pour vous. Une fois que vous avez créé une URL de fonction, son point de terminaison URL ne change jamais. Les points de terminaison URL de fonction ont le format suivant :

```
https://<url-id>.lambda-url.<region>.on.aws
```

> **Note**
>
> URLs Les fonctions ne sont pas prises en charge dans les pays suivants Régions AWS : Asie-Pacifique (Hyderabad`ap-south-2`) (), Asie-Pacifique (Malaisie`ap-southeast-4`) (), Asie-Pacifique (Nouvelle Zélande`ap-southeast-5`) (), Asie-Pacifique (Thaïlande) (`ap-southeast-6`), Asie-Pacifique (Thaïlande) (), Asie-Pacifique (Taipei`ap-southeast-7`) ()`ap-east-2`, Canada Ouest (Calgary) (), Europe (Espagne) ()`ca-west-1`, Europe (Zurich`eu-south-2`) (), Israël (Tel Aviv ()`eu-central-2`, et Moyen-Orient (Émirats arabes unis`il-central-1`) (). `me-central-1`

URLs Les fonctions sont activées par double empilage, supportant et. IPv4 IPv6 Après avoir configuré une URL de fonction pour votre fonction, vous pouvez invoquer cette fonction via son point de terminaison HTTP(S), via un navigateur web, curl, Postman ou n'importe quel client HTTP.

> **Note**
>
> Vous pouvez accéder à l'URL de votre fonction via l'Internet public uniquement. Bien que les fonctions Lambda soient compatibles AWS PrivateLink, les fonctions ne le sont URLs pas.

La fonction Lambda URLs utilise des politiques basées sur les ressources pour la sécurité et le contrôle d'accès. La fonction prend URLs également en charge les options de configuration de partage de ressources entre origines (CORS).

Vous pouvez appliquer une fonction URLs à n'importe quel alias de fonction ou à la version de fonction `$LATEST` non publiée. Vous ne pouvez pas ajouter d'URL de fonction à une autre version de fonction.

La section suivante explique comment créer et gérer une URL de fonction à l'aide de la console Lambda et du AWS CLI modèle CloudFormation

Rubriques

- Création d'une URL de fonction (console)
- Création d'URL de fonction (AWS CLI)
- Ajouter une URL de fonction à un CloudFormation modèle
- Partage des ressources cross-origin (CORS)
- Fonction d'étranglement URLs
- Fonction de désactivation URLs
- Fonction de suppression URLs
- Contrôler l'accès à la fonction Lambda URLs
- Invocation d'URL de fonctions Lambda
- Surveillance des URL de fonctions Lambda
- Sélection d'une méthode pour invoquer votre fonction Lambda à l'aide d'une requête HTTP
- Didacticiel : création d'un point de terminaison webhook à l'aide d'une URL de fonction Lambda

Création d'une URL de fonction (console)

Suivez ces étapes pour créer une URL de fonction à l'aide de la console.

Créer une URL de fonction pour une fonction existante

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom de la fonction pour laquelle vous souhaitez créer l'URL de fonction.
3. Choisissez l'onglet Configuration, puis Function URL (URL de fonction).
4. Choisissez Create function URL (Créer une URL de fonction).
5. Pour le type d'authentification, choisissez AWS_IAMNONE. Pour plus d'informations sur l'authentification de l'URL de fonction, consultez Contrôle d'accès.
6. (Facultatif) Sélectionnez Configure cross-origin resource sharing (CORS) (Configurer le partage des ressources cross-origin [CORS]), puis configurez les paramètres CORS pour l'URL de votre fonction. Pour plus d'informations sur le CORS, consultez Partage des ressources cross-origin (CORS).
7. Choisissez Save (Enregistrer).

Une URL de fonction est créée pour la version non publiée `$LATEST` de votre fonction. L'URL de la fonction s'affiche dans la section Function overview (Présentation de la fonction) de la console.

Créer une URL de fonction pour un alias existant

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom de la fonction avec l'alias pour lequel vous souhaitez créer l'URL de fonction.
3. Choisissez l'onglet Aliases (Alias), puis le nom de l'alias pour lequel vous souhaitez créer l'URL de fonction.
4. Choisissez l'onglet Configuration, puis Function URL (URL de fonction).
5. Choisissez Create function URL (Créer une URL de fonction).
6. Pour le type d'authentification, choisissez AWS_IAMNONE. Pour plus d'informations sur l'authentification de l'URL de fonction, consultez Contrôle d'accès.
7. (Facultatif) Sélectionnez Configure cross-origin resource sharing (CORS) (Configurer le partage des ressources cross-origin [CORS]), puis configurez les paramètres CORS pour l'URL de votre fonction. Pour plus d'informations sur le CORS, consultez Partage des ressources cross-origin (CORS).
8. Choisissez Enregistrer.

Cela crée une URL de fonction pour votre alias de fonction. L'URL de fonction s'affiche dans la section Function overview (Présentation des fonctions) de la console.

Créer une nouvelle fonction avec une URL de fonction

Pour créer une nouvelle fonction avec une URL de fonction (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez Créer une fonction.
3. Sous Basic information (Informations de base), procédez comme suit :
    a. Dans Function name (nom de la fonction), saisissez un nom pour votre fonction, par exemple **`my-function`**.
    b. Pour Runtime, choisissez le langage d'exécution que vous préférez, tel que Node.js 24.
    c. Pour Architecture, choisissez x86_64 ou arm64.
    d. Développez Permissions (Autorisations), puis choisissez si vous souhaitez créer un nouveau rôle d'exécution ou utiliser un rôle existant.
4. Développez Advanced settings (Paramètres avancés), puis sélectionnez Function URL (URL de fonction).
5. Pour le type d'authentification, choisissez AWS_IAMNONE. Pour plus d'informations sur l'authentification de l'URL de fonction, consultez Contrôle d'accès.
6. (Facultatif) Sélectionnez Configure cross-origin resource sharing (CORS) (Configuration du partage des ressources cross-origin). En sélectionnant cette option lors de la création de la fonction, l'URL de votre fonction autorise par défaut les demandes de toutes les origines. Vous pouvez modifier les paramètres CORS de l'URL de votre fonction après avoir créé la fonction. Pour plus d'informations sur le CORS, consultez Partage des ressources cross-origin (CORS).
7. Sélectionnez Create function (Créer une fonction).

Ceci crée une nouvelle fonction avec une URL de fonction pour la `$LATEST` version non publiée de la fonction. L'URL de la fonction s'affiche dans la section Function overview (Présentation des fonctions) de la console.

Création d'URL de fonction (AWS CLI)

Pour créer une URL de fonction pour une fonction Lambda existante à l'aide de AWS Command Line Interface (AWS CLI), exécutez la commande suivante :

```
aws lambda create-function-url-config \
    --function-name my-function \
    --qualifier prod \ // optional
    --auth-type AWS_IAM
    --cors-config {AllowOrigins="https://example.com"} // optional
```

Une URL de fonction est ajoutée au qualificateur **`prod`** pour la fonction **`my-function`**. Pour plus d'informations sur ces paramètres de configuration, consultez CreateFunctionUrlConfig dans la référence d'API.

> **Note**
>
> Pour créer une URL de fonction via le AWS CLI, la fonction doit déjà exister.

Ajouter une URL de fonction à un CloudFormation modèle

Pour ajouter une `AWS::Lambda::Url` ressource à votre CloudFormation modèle, utilisez la syntaxe suivante :

JSON

```
{
  "Type" : "AWS::Lambda::Url",
  "Properties" : {
      "AuthType" : String,
      "Cors" : Cors,
      "Qualifier" : String,
      "TargetFunctionArn" : String
    }
}
```

YAML

```
Type: AWS::Lambda::Url
Properties:
  AuthType: String
  Cors:
    Cors
  Qualifier: String
```

```
  TargetFunctionArn: String
```

Parameters

- (Obligatoire) `AuthType` – Définit le type d'authentification pour l'URL de votre fonction. Les valeurs possibles sont soit `AWS_IAM`, soit `NONE`. Pour limiter l'accès aux utilisateurs authentifiés uniquement, définissez la valeur sur `AWS_IAM`. Pour contourner l'authentification IAM et autoriser n'importe quel utilisateur à envoyer des demandes à votre fonction, définissez la valeur `NONE`.
- (Facultatif) `Cors` – Définit les paramètres CORS de l'URL de votre fonction. Pour ajouter `Cors` à votre `AWS::Lambda::Url` ressource CloudFormation, utilisez la syntaxe suivante.

  Example AWS::Lambda::Url.Cors (JSON)

  ```
  {
    "AllowCredentials" : Boolean,
    "AllowHeaders" : [ String, ... ],
    "AllowMethods" : [ String, ... ],
    "AllowOrigins" : [ String, ... ],
    "ExposeHeaders" : [ String, ... ],
    "MaxAge" : Integer
  }
  ```

  Example AWS::Lambda::Url.Cors (YAML)

  ```
    AllowCredentials: Boolean
    AllowHeaders:
      - String
    AllowMethods:
      - String
    AllowOrigins:
      - String
    ExposeHeaders:
      - String
    MaxAge: Integer
  ```

- (Facultatif) `Qualifier` – Le nom de l'alias.
- (Obligatoire) `TargetFunctionArn` – Le nom ou l'ARN (Amazon Resource Name) de la fonction Lambda. Les formats de nom valides sont notamment les suivants :
  - Nom de fonction – `my-function`

- ARN de fonction – `arn:aws:lambda:us-west-2:123456789012:function:my-function`
- ARN partiel – `123456789012:function:my-function`

Partage des ressources cross-origin (CORS)

Pour définir comment différentes origines peuvent accéder à l'URL de votre fonction, utilisez le partage de ressources cross-origin (CORS). Nous vous recommandons de configurer CORS si vous avez l'intention d'appeler l'URL de votre fonction à partir d'un autre domaine. Lambda prend en charge les en-têtes CORS suivants pour les fonctions. URLs

En-têtes CORS	Propriété de configuration CORS	Exemples de valeur

Access-Control-Allow-Origin	`AllowOrigins`	`*` (autoriser toutes les origines)
		`https://www.example.com`
		`http://localhost:60905`

Access-Control-Allow-Methods	`AllowMethods`	`GET`, `POST`, `DELETE`, `*`
Access-Control-Allow-Headers	`AllowHeaders`	`Date`, `Keep-Alive` , `X-Custom-Header`
Access-Control-Expose-Headers	`ExposeHeaders`	`Date`, `Keep-Alive` , `X-Custom-Header`
Access-Control-Allow-Credentials	`AllowCredentials`	`TRUE`
Access-Control-Max-Age	`MaxAge`	`5` (par défaut), `300`

Lorsque vous configurez CORS pour une URL de fonction à l'aide de la console Lambda ou du AWS CLI, Lambda ajoute automatiquement les en-têtes CORS à toutes les réponses via l'URL de la fonction. Vous pouvez également ajouter manuellement des en-têtes CORS à la réponse de votre fonction. En cas de conflit d'en-têtes, le comportement attendu dépend du type de la requête :

- Pour les requêtes en amont, telles que les requêtes OPTIONS, les en-têtes CORS configurés sur l'URL de la fonction ont priorité. Lambda renvoie uniquement ces en-têtes CORS dans la réponse.
- Pour les requêtes autres que celles effectuées en amont, telles que les requêtes GET ou POST, Lambda renvoie à la fois les en-têtes CORS configurés sur l'URL de la fonction, ainsi que les en-têtes CORS renvoyés par la fonction. Cela peut entraîner une duplication d'en-têtes CORS dans la réponse. Vous pouvez voir une erreur similaire à ce qui suit : `The 'Access-Control-Allow-Origin' header contains multiple values '*, *', but only one is allowed.`

En général, nous recommandons de configurer tous les paramètres CORS sur l'URL de la fonction, plutôt que d'envoyer les en-têtes CORS manuellement dans la réponse de la fonction.

Fonction d'étranglement URLs

La limitation limite le débit auquel la fonction traite les demandes. Cette fonction est utile dans de nombreuses situations, notamment pour empêcher votre fonction de surcharger les ressources en aval ou pour gérer une augmentation soudaine des demandes.

Vous pouvez limiter le débit des demandes traitées par votre fonction Lambda via une URL de fonction en configurant la simultanéité réservée. La simultanéité réservée limite le nombre maximal d'invocations simultanées pour votre fonction. Le débit de demandes maximal par seconde (RPS) de votre fonction est équivalent à 10 fois la simultanéité réservée configurée. Par exemple, si vous configurez votre fonction avec une simultanéité réservée de 100, le RPS maximal est de 1 000.

Lorsque la simultanéité de la fonction dépasse la simultanéité réservée, l'URL de votre fonction renvoie un code d'état HTTP `429`. Si votre fonction reçoit une demande qui dépasse le maximum de 10 fois le RPS par rapport à la simultanéité réservée configurée, vous recevez également une erreur HTTP `429`. Pour de plus amples informations sur la simultanéité réservée, consultez Configuration de la simultanéité réservée pour une fonction.

Fonction de désactivation URLs

En cas d'urgence, vous pouvez rejeter tout le trafic vers l'URL de fonction. Pour désactiver votre URL de fonction, définissez la simultanéité réservée sur zéro. Cela limite toutes les demandes envoyées à

l'URL de votre fonction, ce qui entraîne des réponses d'état HTTP 429. Pour réactiver l'URL de votre fonction, supprimez la configuration de simultanéité réservée ou définissez la configuration sur un montant supérieur à zéro.

Fonction de suppression URLs

Lorsque vous supprimez une URL de fonction, vous ne pouvez pas la récupérer. La création d'une nouvelle URL de fonction entraînera une adresse URL différente.

> **Note**
>
> Si vous supprimez une URL de fonction avec un type d'authentification `NONE`, Lambda ne supprime pas automatiquement la politique basée sur les ressources associée. Si vous souhaitez supprimer cette politique, vous devez le faire manuellement.

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom de la fonction.
3. Choisissez l'onglet Configuration, puis Function URL (URL de fonction).
4. Sélectionnez Delete (Supprimer).
5. Entrez le mot delete dans le champ pour confirmer la suppression.
6. Sélectionnez Delete (Supprimer).

> **Note**
>
> Lorsque vous supprimez une fonction qui possède une URL de fonction, Lambda supprime cette URL de manière asynchrone. Si vous créez immédiatement une nouvelle fonction portant le même nom dans le même compte, il est possible que l'URL de la fonction d'origine soit mappée à la nouvelle fonction au lieu d'être supprimée.

Contrôler l'accès à la fonction Lambda URLs

> **Note**
>
> À compter d'octobre 2025, la nouvelle fonction URLs nécessitera à la fois des `lambda:InvokeFunction` autorisations `lambda:InvokeFunctionUrl` et des autorisations.

Vous pouvez contrôler l'accès à votre fonction Lambda à URLs l'aide du AuthTypeparamètre combiné aux politiques basées sur les ressources associées à votre fonction spécifique. La configuration de ces deux composants détermine qui peut invoquer ou exécuter d'autres actions administratives sur l'URL de votre fonction.

Le paramètre `AuthType` détermine comment Lambda authentifie ou autorise les demandes vers l'URL de votre fonction. Lorsque vous configurez l'URL de votre fonction, vous devez spécifier l'une des options `AuthType` suivantes :

- `AWS_IAM`— Lambda utilise Gestion des identités et des accès AWS (IAM) pour authentifier et autoriser les demandes en fonction de la politique d'identité du principal IAM et de la politique basée sur les ressources de la fonction. Choisissez cette option si vous souhaitez que seuls les utilisateurs et les rôles authentifiés invoquent votre fonction via l'URL de la fonction.

- `NONE` – Lambda n'effectue aucune authentification avant d'invoquer votre fonction. Toutefois, la stratégie basée sur les ressources de votre fonction est toujours en vigueur et doit accorder un accès public avant que l'URL de votre fonction puisse recevoir des demandes. Choisissez cette option pour autoriser un accès public et non authentifié à l'URL de votre fonction.

Pour obtenir des informations supplémentaires sur la sécurité, vous pouvez AWS Identity and Access Management Access Analyzer obtenir une analyse complète de l'accès externe à l'URL de votre fonction. IAM Access Analyzer surveille également les autorisations nouvelles ou mises à jour sur vos fonctions Lambda afin de vous aider à identifier les autorisations accordant un accès public et entre comptes. Vous pouvez utiliser IAM Access Analyzer sans frais. Pour démarrer avec IAM Access Analyzer, consultez Utilisation d' AWS IAM Access Analyzer.

Cette page contient des exemples de politiques basées sur les ressources pour les deux types d'authentification et explique comment créer ces politiques à l'aide de l'opération AddPermissionAPI

ou de la console Lambda. Pour plus d'informations sur la procédure d'invocation de l'URL de votre fonction après avoir configuré les autorisations, consultez Invocation d'URL de fonctions Lambda.

Rubriques

- Utilisation du type d'authentification AWS_IAM
- Utilisation du type d'authentification NONE
- Gouvernance et contrôle d'accès

Utilisation du type d'authentification **AWS_IAM**

Si vous choisissez le type d'authentification `AWS_IAM`, les utilisateurs qui ont besoin d'invoquer l'URL de votre fonction Lambda doivent avoir les autorisations `lambda:InvokeFunctionUrl` et `lambda:InvokeFunction`. Selon la personne qui effectue la demande d'invocation, vous devrez peut-être accorder cette autorisation à l'aide d'une stratégie basée sur les ressources.

Si le principal qui fait la demande se trouve dans la même URL Compte AWS que l'URL de la fonction, il doit soit disposer `lambda:InvokeFunctionUrl` d'`lambda:InvokeFunction`autorisations dans sa politique basée sur l'identité, soit avoir des autorisations qui lui sont accordées dans le cadre de la politique basée sur les ressources de la fonction. En d'autres termes, une stratégie basée sur les ressources est facultative si l'utilisateur possède déjà des autorisations `lambda:InvokeFunctionUrl` et `lambda:InvokeFunction` dans sa stratégie basée sur l'identité. L'évaluation des politiques suit les règles décrites dans la Logique d'évaluation des politiques.

Si le principal qui fait la demande se trouve dans un autre compte, il doit avoir à la fois une stratégie basée sur l'identité qui lui donne des autorisations `lambda:InvokeFunctionUrl` et `lambda:InvokeFunction`, et des autorisations qui lui sont accordées dans une stratégie basée sur les ressources de la fonction qu'il essaie d'invoquer. L'évaluation des stratégies suit les règles décrites dans Comment déterminer si une demande d'accès entre comptes est autorisée.

La politique basée sur les ressources suivante permet au `example` rôle in d'invoquer l'URL Compte AWS `444455556666` de la fonction associée à la fonction. `my-function` La touche de InvokedViaFunctionUrl contexte lambda : limite l'`lambda:InvokeFunction`action aux appels d'URL de fonction. Cela signifie que le principal doit utiliser l'URL de la fonction pour invoquer la fonction. Si vous n'incluez pas `lambda:InvokedViaFunctionUrl`, le principal peut invoquer votre fonction via d'autres méthodes d'invocation, en plus de l'URL de la fonction.

Example – Stratégie intercompte basée sur une ressource

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::444455556666:role/example"
      },
      "Action": "lambda:InvokeFunctionUrl",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:my-function",
      "Condition": {
        "StringEquals": {
          "lambda:FunctionUrlAuthType": "AWS_IAM"
        }
      }
    },
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::444455556666:role/example"
      },
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:my-function",
      "Condition": {
        "Bool": {
          "lambda:InvokedViaFunctionUrl": "true"
        }
      }
    }
  ]
}
```

Vous pouvez créer cette politique basée sur les ressources via la console en suivant les étapes suivantes :

Comment accorder des autorisations d'invocation d'URL à un autre compte (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom de la fonction pour laquelle vous souhaitez accorder des autorisations d'invocation d'URL.
3. Choisissez l'onglet Configuration, puis Permissions (Autorisations).
4. Sous Resource-based policy (stratégie basée sur une ressource), choisissez Add permissions (Ajouter des autorisations).
5. Choisissez Function URL (URL de fonction).
6. Pour le type d'authentification, choisissez AWS_IAM.
7. Entrez un ID de déclaration pour votre déclaration de politique.
8. Pour Principal, saisissez l'ID de compte ou l'Amazon Resource Name (ARN) de l'utilisateur ou du rôle auquel vous souhaitez accorder des autorisations. Par exemple : **444455556666**.
9. Choisissez Enregistrer.

Vous pouvez également créer cette politique à l'aide des commandes add permission AWS Command Line Interface (AWS CLI) suivantes. Lorsque vous utilisez le AWS CLI, vous devez ajouter les `lambda:InvokeFunction` instructions `lambda:InvokeFunctionUrl` et séparément. Par exemple :

```
aws lambda add-permission --function-name my-function \
  --statement-id UrlPolicyInvokeURL \
  --action lambda:InvokeFunctionUrl \
  --principal 444455556666 \
  --function-url-auth-type AWS_IAM
```

```
aws lambda add-permission --function-name my-function \
  --statement-id UrlPolicyInvokeFunction \
  --action lambda:InvokeFunction \
  --principal 444455556666 \
  --invoked-via-function-url
```

Utilisation du type d'authentification **NONE**

> **Important**
>
> Lorsque le type d'authentification de votre URL de fonction est `NONE` et que vous disposez d'une politique basée sur les ressources qui accorde un accès public, tout utilisateur non authentifié ayant de votre URL de fonction peut invoquer votre fonction.

Dans certains cas, vous pouvez souhaiter que votre URL de fonction soit publique. Vus pourriez souhaiter répondre aux demandes envoyées directement à partir d'un navigateur Web. Pour autoriser l'accès public à votre URL de fonction, choisissez le type d'authentification `NONE`.

Si vous choisissez le `NONE` type d'authentification, Lambda n'utilise pas IAM pour authentifier les demandes vers votre URL de fonction. Toutefois, votre fonction doit disposer d'une politique basée sur les ressources qui autorise `lambda:InvokeFunctionUrl` et `lambda:InvokeFunction`. Lorsque vous créez une URL de fonction avec un type d'authentification à `NONE` l'aide de la console ou AWS Serverless Application Model (AWS SAM), Lambda crée automatiquement la politique basée sur les ressources pour vous. Si vous utilisez directement l'API AWS CLI AWS CloudFormation, ou l'API Lambda, vous devez ajouter la politique vous-même.

Nous vous recommandons d'inclure la clé de InvokedViaFunctionUrl contexte lambda : dans vos politiques basées sur les ressources lorsque vous utilisez le `NONE` type d'authentification. Cette clé de contexte garantit que la fonction ne peut être invoquée que via l'URL de la fonction et non via d'autres méthodes d'invocation.

Notez ce qui suit concernant cette politique :

- Toutes les entités peuvent appeler `lambda:InvokeFunctionUrl` et `lambda:InvokeFunction`. Cela signifie que toute personne disposant de votre URL de fonction peut invoquer votre fonction.
- La valeur de la clé de condition `lambda:FunctionUrlAuthType` est `NONE`. Cela signifie que la déclaration de stratégie n'autorise l'accès que lorsque le type d'authentification de votre URL de fonction est également `NONE`.
- La condition `lambda:InvokedViaFunctionUrl` garantit que la fonction ne peut être invoquée que via l'URL de la fonction et non via d'autres méthodes d'invocation.

Example – Politique basée sur les ressources par défaut pour le type d'authentification NONE

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "FunctionURLAllowPublicAccess",
      "Effect": "Allow",
      "Principal": "*",
      "Action": "lambda:InvokeFunctionUrl",
      "Resource": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
      "Condition": {
        "StringEquals": {
          "lambda:FunctionUrlAuthType": "NONE"
        }
      }
    },
    {
      "Sid": "FunctionURLInvokeAllowPublicAccess",
      "Effect": "Allow",
      "Principal": "*",
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
      "Condition": {
        "Bool": {
          "lambda:InvokedViaFunctionUrl": "true"
        }
      }
    }
  ]
}
```

Créez la politique basée sur les ressources à l'aide du AWS CLI

À moins d'utiliser la console ou AWS SAM de créer une URL de fonction avec un type d'authentification`NONE`, vous devez ajouter vous-même la politique basée sur les ressources. Utilisez les commandes suivantes pour créer des instructions pour les autorisations `lambda:InvokeFunctionUrl` et `lambda:InvokeFunction`. Chaque instruction doit être ajoutée dans une commande distincte.

```
aws lambda add-permission \
  --function-name UrlTestFunction \
  --statement-id UrlPolicyInvokeURL \
  --action lambda:InvokeFunctionUrl \
  --principal * \
  --function-url-auth-type NONE
```

```
aws lambda add-permission \
  --function-name UrlTestFunction \
  --statement-id UrlPolicyInvokeFunction \
  --action lambda:InvokeFunction \
  --principal * \
  --invoked-via-function-url
```

> **Note**
>
> Si vous supprimez une URL de fonction avec un type d'authentification `NONE`, Lambda ne supprime pas automatiquement la politique basée sur les ressources associée. Si vous souhaitez supprimer cette politique, vous devez le faire manuellement.

Si la stratégie basée sur les ressources d'une fonction n'accorde pas les autorisations `lambda:invokeFunctionUrl` et `lambda:InvokeFunction`, les utilisateurs obtiendront alors un code d'erreur 403 Forbidden lorsqu'ils essaient d'invoquer l'URL de votre fonction. Cela se produira même si l'URL de la fonction utilise le type d'authentification `NONE`.

Gouvernance et contrôle d'accès

Outre les autorisations d'appel d'URL de fonction, vous pouvez également contrôler l'accès aux actions utilisées pour configurer la fonction URLs. Lambda prend en charge les actions de politique IAM suivantes pour les fonctions : URLs

- `lambda:InvokeFunctionUrl` – Invoquer une fonction Lambda à l'aide de l'URL de fonction.
- `lambda:CreateFunctionUrlConfig` – Créer une URL de fonction et définir son `AuthType`.
- `lambda:UpdateFunctionUrlConfig` – Mettre à jour la configuration d'une URL de fonction et son `AuthType`.
- `lambda:GetFunctionUrlConfig` – Affichez les détails d'une URL de fonction.
- `lambda:ListFunctionUrlConfigs` – Répertorier les configurations d'URL de fonction.

- `lambda:DeleteFunctionUrlConfig` – Supprimer une URL de fonction.

Pour autoriser ou refuser l'accès à l'URL des fonctions à d'autres AWS entités, incluez ces actions dans les politiques IAM. Par exemple, la politique suivante accorde au `example` rôle dans les Compte AWS `444455556666` autorisations de mettre à jour l'URL de la fonction pour la fonction **`my-function`** dans le compte`123456789012`.

Example exemple de stratégie d'URL de fonction inter-comptes

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::444455556666:role/example"
            },
            "Action": "lambda:UpdateFunctionUrlConfig",
            "Resource": "arn:aws:lambda:us-east-2:123456789012:function:my-
function"
        }
    ]
}
```

Clés de condition

Pour un contrôle d'accès précis à votre fonction URLs, utilisez les touches contextuelles de condition. Lambda prend en charge les clés de contexte suivantes pour la fonction : URLs

- `lambda:FunctionUrlAuthType` : définit une valeur d'énumération décrivant le type d'authentification utilisé par l'URL de votre fonction. La valeur peut être `AWS_IAM` ou `NONE`.
- `lambda:InvokedViaFunctionUrl` : limite l'action `lambda:InvokeFunction` aux appels effectués via l'URL de la fonction. Cela garantit que la fonction ne peut être invoquée que via l'URL de la fonction et non via d'autres méthodes d'invocation. Pour des exemples de politiques basées sur les ressources qui utilisent la clé de contexte `lambda:InvokedViaFunctionUrl`,

consultez les exemples dans Utilisation du type d'authentification `AWS_IAM` et Utilisation du type d'authentification `NONE`.

Vous pouvez utiliser ces clés de contexte dans les stratégies associées à votre fonction. Par exemple, vous souhaiterez peut-être limiter les personnes autorisées à modifier la configuration de votre fonction URLs. Pour rejeter toutes les demandes `UpdateFunctionUrlConfig` à n'importe quelle fonction avec un type d'authentification URL `NONE`, vous pouvez définir la stratégie suivante :

Example exemple de stratégie d'URL de fonction avec refus explicite

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Principal": "*",
            "Action":[
                "lambda:UpdateFunctionUrlConfig"
            ],
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:*",
            "Condition": {
                "StringEquals": {
                    "lambda:FunctionUrlAuthType": "NONE"
                }
            }
        }
    ]
}
```

Pour accorder le `example` rôle dans Compte AWS `444455556666` les autorisations de création `CreateFunctionUrlConfig` et de `UpdateFunctionUrlConfig` requêtes sur les fonctions avec un type d'authentification URL`AWS_IAM`, vous pouvez définir la politique suivante :

Example exemple de stratégie d'URL de fonction avec autorisation explicite

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::444455556666:role/example"
            },
            "Action":[
                "lambda:CreateFunctionUrlConfig",
                "lambda:UpdateFunctionUrlConfig"
            ],
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:*",
            "Condition": {
                "StringEquals": {
                    "lambda:FunctionUrlAuthType": "AWS_IAM"
                }
            }
        }
    ]
}
```

Vous pouvez également utiliser cette clé de condition dans une stratégie de contrôle de service (SCP). SCPs À utiliser pour gérer les autorisations dans l'ensemble d'une organisation dans AWS Organizations. Par exemple, pour empêcher les utilisateurs de créer ou de mettre à jour des fonctions URLs utilisant autre chose que le type d'`AWS_IAM`authentification, appliquez la politique de contrôle des services suivante :

Example exemple d'URL de fonction SCP avec refus explicite

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
```

```
            "Effect": "Deny",
            "Action":[
                "lambda:CreateFunctionUrlConfig",
                "lambda:UpdateFunctionUrlConfig"
            ],
            "Resource": "arn:aws:lambda:*:123456789012:function:*",
            "Condition": {
                "StringNotEquals": {
                    "lambda:FunctionUrlAuthType": "AWS_IAM"
                }
            }
        }
    ]
}
```

Invocation d'URL de fonctions Lambda

Une URL de fonction est un point de terminaison HTTP(S) dédié pour votre fonction Lambda. Vous pouvez créer et configurer une URL de fonction via la console Lambda ou l'API Lambda.

> **Tip**
>
> Lambda propose deux méthodes pour invoquer votre fonction via un point de terminaison HTTP : les URL de fonction et Amazon API Gateway. Si vous ne savez pas quelle est la meilleure méthode pour votre cas d'utilisation, consultez the section called " URLs Function et Amazon API Gateway".

Lorsque vous créez une URL de fonction, Lambda génère automatiquement un point de terminaison URL unique pour vous. Une fois que vous avez créé une URL de fonction, son point de terminaison URL ne change jamais. Les points de terminaison URL de fonction ont le format suivant :

```
https://<url-id>.lambda-url.<region>.on.aws
```

> **Note**
>
> Les URL de fonction ne sont pas prises en charge dans les Régions AWS suivantes : Asie-Pacifique (Hyderabad) (`ap-south-2`), Asie-Pacifique (Melbourne) (`ap-southeast-4`), Asie-Pacifique (Malaisie) (`ap-southeast-5`), Asie-Pacifique (Nouvelle-Zélande) (`ap-southeast-6`), Asie-Pacifique (Thaïlande) (`ap-southeast-7`), Asie-Pacifique (Taipei) (`ap-east-2`), Canada Ouest (Calgary) (`ca-west-1`), Europe (Espagne) (`eu-south-2`), Europe (Zurich) (`eu-central-2`), Israël (Tel Aviv) (`il-central-1`) et Moyen-Orient (EAU) (`me-central-1`).

Les URL de fonctions sont compatibles avec la pile double, prenant en charge les adresses IPv4 et IPv6. Après avoir configuré votre URL de fonction, vous pouvez invoquer votre fonction à travers son point de terminaison HTTP(S) via un navigateur Web, curl, Postman, ou n'importe quel client HTTP. Pour invoquer l'URL d'une fonction, vous devez avoir les autorisations `lambda:InvokeFunctionUrl` et `lambda:InvokeFunction`. Pour de plus amples informations, consultez Contrôle d'accès.

Rubriques

- Principes de base de l'invocation d'une URL de fonction
- Charges utiles de demandes et de réponses

Principes de base de l'invocation d'une URL de fonction

Si votre URL de votre fonction utilise le type d'authentification `AWS_IAM`, vous devez signer chaque demande HTTP en utilisant AWSSignature Version 4 (SigV4). Des outils comme Postman offrent des moyens intégrés de signer vos demandes avec SigV4.

Si vous n'utilisez pas d'outil pour signer des demandes HTTP sur l'URL de votre fonction, vous devez signer manuellement chaque demande à l'aide de SigV4. Lorsque l'URL de votre fonction reçoit une demande, Lambda calcule également la signature SigV4. Lambda traite la demande uniquement si les signatures correspondent. Pour obtenir des instructions sur la signature manuelle de vos demandes avec SigV4, consultez Signature de demandes AWS avec Signature Version 4 dans le Guide Référence générale d'Amazon Web Services.

Si l'URL de votre fonction utilise le type d'authentification `NONE`, vous n'avez pas besoin de signer vos demandes avec SigV4. Vous pouvez invoquer votre fonction via un navigateur Web, curl, Postman ou n'importe quel client HTTP.

Pour tester des demandes `GET` simples vers votre fonction, utilisez un navigateur web. Par exemple, si l'URL de votre fonction est `https://abcdefg.lambda-url.us-east-1.on.aws`, et si elle prend en compte un paramètre de chaîne `message`, l'URL de votre demande peut se présenter comme suit :

```
https://abcdefg.lambda-url.us-east-1.on.aws/?message=HelloWorld
```

Pour tester d'autres demandes HTTP, telles qu'une demande `POST`, vous pouvez utiliser un outil tel que curl. Par exemple, si vous souhaitez inclure certaines données JSON dans une demande `POST` pour l'URL de votre fonction, vous pouvez utiliser la commande curl suivante :

```
curl -v 'https://abcdefg.lambda-url.us-east-1.on.aws/?message=HelloWorld' \
-H 'content-type: application/json' \
-d '{ "example": "test" }'
```

Charges utiles de demandes et de réponses

Lorsqu'un client appelle l'URL de votre fonction, Lambda mappe la demande sur un objet événement avant de la transmettre à votre fonction. La réponse de votre fonction est ensuite mappée à une réponse HTTP que Lambda renvoie au client via l'URL de la fonction.

Les formats d'événements de demande et de réponse suivent le même schéma que le format de charge utile version 2.0 d'Amazon API Gateway.

Format de la charge utile de demande

Une charge utile de demande a la structure suivante :

```
{
  "version": "2.0",
  "routeKey": "$default",
  "rawPath": "/my/path",
  "rawQueryString": "parameter1=value1&parameter1=value2&parameter2=value",
  "cookies": [
    "cookie1",
    "cookie2"
  ],
  "headers": {
    "header1": "value1",
    "header2": "value1,value2"
  },
  "queryStringParameters": {
    "parameter1": "value1,value2",
    "parameter2": "value"
  },
  "requestContext": {
    "accountId": "123456789012",
    "apiId": "<urlid>",
    "authentication": null,
    "authorizer": {
        "iam": {
                "accessKey": "AKIA...",
                "accountId": "111122223333",
                "callerId": "AIDA...",
                "cognitoIdentity": null,
                "principalOrgId": null,
                "userArn": "arn:aws:iam::111122223333:user/example-user",
                "userId": "AIDA..."
```

```
        }
    },
    "domainName": "<url-id>.lambda-url.us-west-2.on.aws",
    "domainPrefix": "<url-id>",
    "http": {
      "method": "POST",
      "path": "/my/path",
      "protocol": "HTTP/1.1",
      "sourceIp": "123.123.123.123",
      "userAgent": "agent"
    },
    "requestId": "id",
    "routeKey": "$default",
    "stage": "$default",
    "time": "12/Mar/2020:19:03:58 +0000",
    "timeEpoch": 1583348638390
  },
  "body": "Hello from client!",
  "pathParameters": null,
  "isBase64Encoded": false,
  "stageVariables": null
}
```

Paramètre	Description	Exemple
`version`	La version du format de la charge utile pour cet événement. Les URL de fonctions Lambda prennent actuellement en charge le format de charge utile version 2.0.	`2.0`
`routeKey`	Les URL de fonctions n'utilisent pas ce paramètre. Lambda le définit sur `$default` comme espace réservé.	`$default`
`rawPath`	Chemin d'accès de la demande. Par exemple, si	`/example/test/demo`

Paramètre	Description	Exemple
	l'URL de la demande est `https://{url-id}.lambda-url.{region}.on.aws/example/test/demo` , alors la valeur du chemin brut est `/example/test/demo` .	
`rawQueryString`	La chaîne brute contenant les paramètres de la chaîne de la requête de la demande. Les caractères pris en charge sont les suivants : `a-z`, `A-Z`, `0-9`, `.`, `_`, `-`, `%`, &, =, et +.	`"?parameter1=value1&parameter2=value2"`
`cookies`	Un tableau contenant tous les cookies envoyés dans le cadre de la demande.	`["Cookie_1=Value_1", "Cookie_2=Value_2"]`
`headers`	La liste des en-têtes de demande, présentée sous forme de paires valeur-clé.	`{"header1": "value1", "header2": "value2"}`
`queryStringParameters`	Paramètres de requête pour la demande. Par exemple, si l'URL de la demande est `https://{url-id}.lambda-url.{region}.on.aws/example?name=Jane` , alors la valeur `queryStringParameters` est un objet JSON avec une clé `name` et une valeur `Jane`.	`{"name": "Jane"}`

Paramètre	Description	Exemple
`requestContext`	Un objet qui contient des informations supplémentaires sur la demande, comme `requestId` , l'heure de la demande, et l'identité de l'appelant s'il est autorisé via Gestion des identités et des accès AWS (IAM).	
`requestContext.accountId`	L'identifiant Compte AWS du propriétaire de la fonction.	`"123456789012"`
`requestContext.apiId`	L'identifiant de l'URL de la fonction.	`"33anwqw8fj"`
`requestContext.authentication`	Les URL de fonctions n'utilisent pas ce paramètre. Lambda le définit sur `null`.	`null`
`requestContext.authorizer`	Un objet qui contient des informations sur l'identité de l'appelant si l'URL de la fonction utilise le type d'authentification `AWS_IAM`. Sinon, Lambda définit cette valeur sur `null`.	
`requestContext.authorizer.iam.accessKey`	La clé d'accès de l'identité de l'appelant.	`"AKIAIOSFODNN7EXAMPLE"`
`requestContext.authorizer.iam.accountId`	L'identifiant Compte AWS de l'identité de l'appelant.	`"111122223333"`

Paramètre	Description	Exemple
`requestContext.authorizer.iam.callerId`	L'ID (ID d'utilisateur) de l'appelant.	`"AIDACKCEVSQ6C2EXAMPLE"`
`requestContext.authorizer.iam.cognitoIdentity`	Les URL de fonctions n'utilisent pas ce paramètre. Lambda le définit sur `null` ou l'exclut de JSON.	`null`
`requestContext.authorizer.iam.principalOrgId`	L'ID de l'organisation du principal associé à l'ID de l'appelant.	`"AIDACKCEVSQORGEXAMPLE"`
`requestContext.authorizer.iam.userArn`	L'Amazon Resource Name (ARN) utilisateur de l'identité de l'appelant.	`"arn:aws:iam::111122223333:user/example-user"`
`requestContext.authorizer.iam.userId`	L'identifiant utilisateur de l'identité de l'appelant.	`"AIDACOSFODNN7EXAMPLE2"`
`requestContext.domainName`	Le nom de domaine de l'URL de la fonction.	`"<url-id>.lambda-url.us-west-2.on.aws"`
`requestContext.domainPrefix`	Le nom de domaine de l'URL de la fonction.	`"<url-id>"`
`requestContext.http`	Un objet qui contient des informations sur la demande HTTP.	

Paramètre	Description	Exemple
`requestContext.http.method`	La méthode HTTP utilisée dans cette demande. Les valeurs valides sont les suivantes : `GET`, `POST`, `PUT`, `HEAD`, `OPTIONS`, `PATCH` et `DELETE`.	`GET`
`requestContext.http.path`	Chemin d'accès de la demande. Par exemple, si l'URL de la demande est `https://{url-id}.lambda-url.{region}.on.aws/example/test/demo` , alors la valeur du chemin est `/example/test/demo` .	`/example/test/demo`
`requestContext.http.protocol`	Le protocole de la demande.	`HTTP/1.1`
`requestContext.http.sourceIp`	L'adresse IP source de la connexion TCP immédiate qui fait la demande.	`123.123.123.123`
`requestContext.http.userAgent`	La valeur de l'en-tête de la demande Utilisateur-Agent.	`Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) Gecko/20100101 Firefox/42.0`
`requestContext.requestId`	L'identifiant de la demande d'invocation. Vous pouvez utiliser cet ID pour suivre les journaux d'invocation liés à votre fonction.	`e1506fd5-9e7b-434f-bd42-4f8fa224b599`

Paramètre	Description	Exemple
`requestContext.routeKey`	Les URL de fonctions n'utilisent pas ce paramètre. Lambda le définit sur `$default` comme espace réservé.	`$default`
`requestContext.stage`	Les URL de fonctions n'utilisent pas ce paramètre. Lambda le définit sur `$default` comme espace réservé.	`$default`
`requestContext.time`	Horodatage de la demande.	`"07/Sep/2021:22:50:22 +0000"`
`requestContext.timeEpoch`	L'horodatage de la demande, en heure d'époque Unix.	`"1631055022677"`
`body`	Le corps de la demande. Si le type de contenu de la demande est binaire, le corps est codé en base64.	`{"key1": "value1", "key2": "value2"}`
`pathParameters`	Les URL de fonctions n'utilisent pas ce paramètre. Lambda le définit sur `null` ou l'exclut de JSON.	`null`
`isBase64Encoded`	`TRUE` si le corps est une charge utile binaire et codé en base64. Sinon `FALSE`.	`FALSE`
`stageVariables`	Les URL de fonctions n'utilisent pas ce paramètre. Lambda le définit sur `null` ou l'exclut du JSON.	`null`

Format de la charge utile de la réponse

Lorsque votre fonction renvoie une réponse, Lambda analyse la réponse et la convertit en réponse HTTP. Les charges utiles de la réponse de fonction ont le format suivant :

```
{
   "statusCode": 201,
    "headers": {
        "Content-Type": "application/json",
        "My-Custom-Header": "Custom Value"
    },
    "body": "{ \"message\": \"Hello, world!\" }",
    "cookies": [
        "Cookie_1=Value1; Expires=21 Oct 2021 07:48 GMT",
        "Cookie_2=Value2; Max-Age=78000"
    ],
    "isBase64Encoded": false
}
```

Lambda déduit le format de réponse pour vous. Si votre fonction renvoie un JSON valide et ne renvoie pas un `statusCode`, Lambda assume ce qui suit :

- `statusCode` is `200`.

> **Note**
>
> Les valeurs de `statusCode` valides sont comprises entre 100 et 599.

- `content-type` is `application/json`.
- `body` est la réponse de la fonction.
- `isBase64Encoded` is `false`.

Les exemples suivants montrent comment la sortie de votre fonction Lambda est mappée à la charge utile de réponse et comment la charge utile de réponse est mappée à la réponse HTTP finale. Lorsque le client invoque l'URL de votre fonction, il voit la réponse HTTP.

Exemple de sortie pour une réponse de chaîne

Sortie de la fonction Lambda	Sortie de réponse interprétée	Réponse HTTP (ce que voit le client)

`"Hello, world!"`	`{ "statusCode": 200, "body": "Hello, world!", "headers": { "content-type": "application/json" }, "isBase64Encoded": false }`	`HTTP/2 200`
		`date: Wed, 08 Sep 2021 18:02:24 GMT`
		`content-type: application/json`
		`content-length: 15`
		
		`"Hello, world!"`

Exemple de sortie pour une réponse JSON

Sortie de la fonction Lambda	Sortie de réponse interprétée	Réponse HTTP (ce que voit le client)

`{ "message": "Hello, world!" }`	`{ "statusCode": 200, "body": { "message": "Hello, world!" }, "headers": { "content-type": "application/json" }, "isBase64Encoded": false }`	`HTTP/2 200`
		`date: Wed, 08 Sep 2021 18:02:24 GMT`
		`content-type: application/json`
		`content-length: 34`
		
		`{ "message": "Hello, world!" }`

Exemple de sortie pour une réponse personnalisée

Sortie de la fonction Lambda	Sortie de réponse interprétée	Réponse HTTP (ce que voit le client)

<pre>{
"statusCode": 201,
"headers": {
"Content-Type": "application/json",
"My-Custom-Header": "Custom Value"
},
"body": JSON.stringify({
"message": "Hello, world!"
}),
"isBase64Encoded": false
}</pre>
	<pre>{
	"statusCode": 201,
	"headers": {
	"Content-Type": "application/json",
	"My-Custom-Header": "Custom Value"
	},
	"body": JSON.stringify({
	"message": "Hello, world!"
	}),
	"isBase64Encoded": false
	}</pre>
		<pre>HTTP/2 201
		date: Wed, 08 Sep 2021 18:02:24 GMT
		content-type: application/json
		content-length: 27
		my-custom-header: Custom Value
		
		{
		"message": "Hello, world!"
		}</pre>

Cookies

Pour renvoyer des cookies depuis votre fonction, n'ajoutez pas manuellement des en-têtes `set-cookie`. Incluez plutôt les cookies dans votre objet de charge utile de réponse. Lambda les interprète automatiquement et les ajoute comme en-têtes `set-cookie` de votre réponse HTTP, comme dans l'exemple suivant.

Sortie de la fonction Lambda	Réponse HTTP (ce que voit le client)

<pre>{
"statusCode": 201,
"headers": {
"Content-Type": "application/json",
"My-Custom-Header": "Custom Value"
},
"body": JSON.stringify({
"message": "Hello, world!"</pre>
	<pre>HTTP/2 201
	date: Wed, 08 Sep 2021 18:02:24 GMT
	content-type: application/json
	content-length: 27
	my-custom-header: Custom Value
	set-cookie: Cookie_1=Value2; Expires=21 Oct 2021 07:48 GMT
	set-cookie: Cookie_2=Value2; Max-Age=78000</pre>

Sortie de la fonction Lambda	Réponse HTTP (ce que voit le client)

<pre>   }),
    "cookies": [
        "Cookie_1=Value1; Expires=21
 Oct 2021 07:48 GMT",
        "Cookie_2=Value2; Max-Age=7
8000"
    ],
    "isBase64Encoded": false
}</pre>
	<pre>{
	  "message": "Hello, world!"
	}</pre>

Surveillance des URL de fonctions Lambda

Vous pouvez utiliser AWS CloudTrail et Amazon CloudWatch pour surveiller vos URL de fonctions.

Rubriques

- URL de fonctions de surveillance avec CloudTrail
- Métriques CloudWatch pour les URL de fonctions

URL de fonctions de surveillance avec CloudTrail

Pour les URL de fonctions, Lambda prend automatiquement en charge la journalisation des opérations API suivantes en tant qu'événements dans des fichiers journaux CloudTrail :

- CreateFunctionUrlConfig
- UpdateFunctionUrlConfig
- DeleteFunctionUrlConfig
- GetFunctionUrlConfig
- ListFunctionUrlConfigs

Chaque entrée de journal contient des informations sur l'identité de l'appelant, le moment où la demande a été faite, et d'autres détails. Vous pouvez voir tous les événements des 90 derniers jours en consultant votre Event history (Historique des événements) CloudTrail. Pour conserver les enregistrements de plus de 90 jours, vous pouvez créer un journal d'activité.

Par défaut, CloudTrail ne journalise pas les demandes `InvokeFunctionUrl`, qui sont considérées comme des événements de données. Cependant, vous pouvez activer la journalisation des événements de données dans CloudTrail. Pour plus d'informations, consultez Consignation d'événements de données pour les journaux d'activité dans le Guide de l'utilisateur AWS CloudTrail.

Métriques CloudWatch pour les URL de fonctions

Lambda envoie des métriques agrégées concernant les demandes d'URL de fonction vers CloudWatch. Avec ces métriques, vous pouvez surveiller les URL de vos fonctions, créer des tableaux de bord et configurer des alarmes dans la console CloudWatch.

Les URL de fonctions prennent en charge les métriques d'invocation suivantes. Nous vous recommandons de consulter ces métriques avec la statistique `Sum`.

- `UrlRequestCount` – Nombre de demandes faites vers cette URL de fonction.
- `Url4xxCount` – Nombre de demandes ayant retourné un code d'état HTTP 4XX. Les codes de la série 4XX indiquent des erreurs côté client, telles que des demandes erronées.
- `Url5xxCount` – Nombre de demandes ayant retourné un code d'état HTTP 5XX. Les codes de la série 5XX indiquent des erreurs côté serveur, telles que des erreurs de fonction et des délais d'attente.

Les URL de fonctions prennent également en charge la métrique de performance suivante. Nous vous recommandons de consulter ces métriques avec les statistiques `Average` ou `Max`.

- `UrlRequestLatency` – Le temps écoulé entre le moment où l'URL de fonction reçoit une demande et celui où l'URL de fonction renvoie une réponse.

Chacune de ces métriques d'invocation et de performance prend en charge les dimensions suivantes :

- `FunctionName` – Affichez les métriques agrégées pour les URL de fonctions attribuées à la version non publiée `$LATEST` d'une fonction, ou à tout alias de la fonction. Par exemple, `hello-world-function`.
- `Resource` – Affichez les métriques d'une URL de fonction spécifique. Cela est défini par un nom de fonction, accompagné de la version non publiée `$LATEST` de la fonction ou d'un alias de la fonction. Par exemple, `hello-world-function:$LATEST`.
- `ExecutedVersion` – Affichez les métriques d'une URL de fonction spécifique en fonction de la version exécutée. Vous pouvez utiliser cette dimension principalement pour suivre l'URL de la fonction affectée à la version non publiée `$LATEST`.

Sélection d’une méthode pour invoquer votre fonction Lambda à l’aide d’une requête HTTP

De nombreux cas d’utilisation courants de Lambda impliquent d’invoquer votre fonction à l’aide d’une requête HTTP. Vous pourriez souhaiter qu’une application Web invoque votre fonction par le biais d’une demande de navigateur. Les fonctions Lambda peuvent également être utilisées pour créer un REST complet APIs, gérer les interactions des utilisateurs à partir d'applications mobiles, traiter des données provenant de services externes via des appels HTTP ou créer des webhooks personnalisés.

Les sections suivantes expliquent quels sont vos choix pour invoquer Lambda via HTTP et fournissent des informations qui vous aideront à prendre la bonne décision pour votre cas d'utilisation particulier.

Options proposées lors de la sélection d'une méthode d'appel HTTP

Lambda propose deux méthodes principales pour appeler une fonction à l'aide d'une requête HTTP : function URLs et API Gateway. Les principales différences entre ces deux options sont les suivantes :

- La fonction Lambda URLs fournit un point de terminaison HTTP simple et direct pour une fonction Lambda. Elles sont optimisées pour la simplicité et la rentabilité et constituent le moyen le plus rapide d'exposer une fonction Lambda via HTTP.
- API Gateway est un service plus avancé permettant de créer des fonctionnalités complètes APIs. API Gateway est optimisé pour créer et gérer des productions APIs à grande échelle et fournit des outils complets pour la sécurité, la surveillance et la gestion du trafic.

Recommandations si vous connaissez déjà vos besoins

Si vous connaissez déjà bien vos besoins, voici nos recommandations de base :

Nous recommandons cette fonction URLs pour les applications simples ou le prototypage où vous n'avez besoin que de méthodes d'authentification et de request/response manipulation de base et où vous souhaitez réduire les coûts et la complexité au minimum.

API Gateway est un meilleur choix pour les applications de production à grande échelle ou pour les cas où vous avez besoin de fonctionnalités plus avancées telles que la prise en charge d'OpenAPI Description, un choix d'options d'authentification, des noms de domaine personnalisés ou une request/response gestion riche, notamment la régulation, la mise en cache et la transformation. request/response

Éléments à prendre en compte lors de la sélection d'une méthode pour invoquer votre fonction Lambda

Lorsque vous choisissez entre function URLs et API Gateway, vous devez prendre en compte les facteurs suivants :

- Vos besoins en matière d'authentification, par exemple si vous avez besoin OAuth d'Amazon Cognito pour authentifier les utilisateurs

- Vos exigences en matière de mise à l'échelle et la complexité de l'API que vous souhaitez implémenter
- Si vous avez besoin de fonctionnalités avancées telles que la validation et le request/response formatage des demandes
- Vos besoins en matière de surveillance
- Vos objectifs en matière de coûts

La compréhension de ces facteurs vous permettra de choisir l'option qui répond le mieux à vos exigences en matière de sécurité, de complexité et de coût.

Le tableau suivant résume les différences entre les deux options.

Authentification

- La fonction URLs fournit des options d'authentification de base via Gestion des identités et des accès AWS (IAM). Vous pouvez configurer vos points de terminaison pour qu'ils soient publics (aucune authentification) ou pour qu'ils nécessitent une authentification IAM. Avec l'authentification IAM, vous pouvez utiliser des AWS informations d'identification standard ou des rôles IAM pour contrôler l'accès. Bien que simple à configurer, cette approche offre des options limitées par rapport aux autres méthodes d'authentification.
- API Gateway donne accès à une gamme plus complète d'options d'authentification. Outre l'authentification IAM, vous pouvez utiliser des autorisateurs Lambda (logique d'authentification personnalisée), des groupes d'utilisateurs Amazon Cognito et des flux .0. OAuth2 Cette flexibilité vous permet de mettre en œuvre des schémas d'authentification complexes, notamment des fournisseurs d'authentification tiers, une authentification basée sur des jetons et une authentification multifactorielle.

Traitement des requêtes et réponses

- La fonction URLs fournit une gestion de base des requêtes et des réponses HTTP. Elles prennent en charge les méthodes HTTP standard et créent une prise en charge intégrée du partage des ressources entre origines (CORS, cross-origin resource sharing). Bien qu'elles puissent gérer naturellement les charges utiles JSON et les paramètres de requête, elles n'offrent pas de fonctionnalités de transformation ou de validation des requêtes. La gestion des réponses est tout aussi simple : le client reçoit la réponse de votre fonction Lambda exactement telle que Lambda la renvoie.

- API Gateway fournit des fonctionnalités sophistiquées de gestion des requêtes et des réponses. Vous pouvez définir des validateurs de demandes, transformer les demandes et les réponses à l'aide de modèles de mappage, configurer request/response des en-têtes et implémenter la mise en cache des réponses. API Gateway prend également en charge les charges utiles binaires et les noms de domaine personnalisés et peut modifier les réponses avant qu'elles n'atteignent le client. Vous pouvez configurer des modèles pour la request/response validation et la transformation à l'aide du schéma JSON.

Mise à l'échelle

- Les fonctions s' URLsadaptent directement aux limites de simultanéité de votre fonction Lambda et gérez les pics de trafic en augmentant la taille de votre fonction jusqu'à sa limite de simultanéité maximale configurée. Une fois cette limite atteinte, Lambda répond aux requêtes supplémentaires avec des réponses HTTP 429. Il n'existe aucun mécanisme de mise en file d'attente intégré. La gestion de la mise à l'échelle dépend donc entièrement de la configuration de votre fonction Lambda. Par défaut, les fonctions Lambda sont limitées à 1 000 exécutions simultanées par fonction. Région AWS
- API Gateway fournit des fonctionnalités de dimensionnement supplémentaires en plus de la propre mise à l'échelle de Lambda. Il inclut des commandes intégrées de mise en file d'attente et de limitation des requêtes, ce qui vous permet de gérer les pics de trafic de manière plus élégante. API Gateway peut traiter jusqu'à 10 000 requêtes par seconde et par région par défaut, avec une capacité de débordement de 5 000 requêtes par seconde. Il fournit également des outils pour limiter les requêtes à différents niveaux (API, étape ou méthode) afin de protéger votre dorsal.

Contrôle

- Les fonctions URLs offrent une surveillance de base via CloudWatch les métriques Amazon, notamment le nombre de demandes, la latence et les taux d'erreur. Vous avez accès aux métriques et aux journaux Lambda standard, qui indiquent les requêtes brutes entrant dans votre fonction. Bien que cela fournisse une visibilité opérationnelle essentielle, les métriques se concentrent principalement sur l'exécution des fonctions.
- API Gateway fournit des fonctionnalités de surveillance complètes, notamment des métriques détaillées, des options de journalisation et de suivi. Vous pouvez surveiller les appels d'API, la latence, les taux d'erreur et hit/miss les taux de cache via CloudWatch. API Gateway s'intègre également au AWS X-Ray traçage distribué et fournit des formats de journalisation personnalisables.

Cost

- Les fonctions URLs suivent le modèle de tarification Lambda standard : vous ne payez que pour les appels de fonctions et le temps de calcul. Il n'y a pas de frais supplémentaires pour le point de terminaison de l'URL lui-même. Cela en fait un choix rentable pour les applications simples APIs ou à faible trafic si vous n'avez pas besoin des fonctionnalités supplémentaires d'API Gateway.
- API Gateway propose un niveau gratuit qui inclut un million d'appels d'API reçus pour REST APIs et un million d'appels d'API reçus pour HTTP APIs. Ensuite, API Gateway facture les appels d'API, le transfert de données et la mise en cache (si activée). Reportez-vous à la page de tarification d'API Gateway pour connaître les coûts associés à votre propre cas d'utilisation.

Autres fonctions

- URLsLes fonctions sont conçues pour la simplicité et l'intégration directe de Lambda. Ils prennent en charge les points de terminaison HTTP et HTTPS, offrent un support CORS intégré et fournissent des points de terminaison à double pile (IPv4 et IPv6). Bien qu'elles ne disposent d'aucune fonctionnalité avancée, elles excellent dans les scénarios où vous avez besoin d'un moyen rapide et simple d'exposer les fonctions Lambda via HTTP.
- API Gateway inclut de nombreuses fonctionnalités supplémentaires telles que la gestion des versions des API, la gestion des étapes, les clés d'API pour les plans d'utilisation, la documentation des API via Swagger/OpenAPI, le mode WebSocket APIs privé au sein d' APIs un VPC et l'intégration WAF pour une sécurité accrue. Il prend également en charge les déploiements Canary, les intégrations fictives à des fins de test et l'intégration avec d'autres solutions que Services AWS Lambda.

Sélection d'une méthode pour invoquer votre fonction Lambda

Maintenant que vous avez pris connaissance des critères de sélection entre la fonction Lambda URLs et API Gateway et des principales différences entre les deux, vous pouvez sélectionner l'option qui répond le mieux à vos besoins et utiliser les ressources suivantes pour vous aider à commencer à l'utiliser.

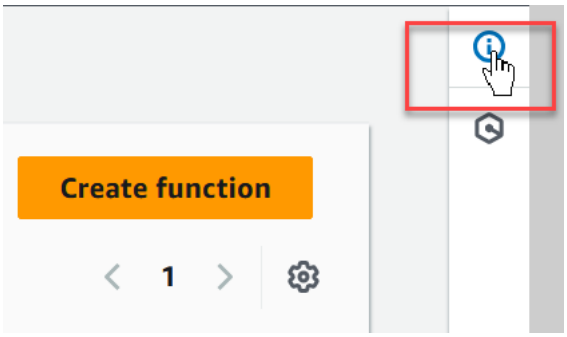
Function URLs

Commencez à utiliser Function URLs grâce aux ressources suivantes

- Suivre le tutoriel Création d'une fonction Lambda avec une URL de fonction

- Pour en savoir plus sur URLs les fonctions, consultez le the section called “Fonction URLs” chapitre de ce guide
- Essayez le tutoriel guidé intégré à la console Créer une application Web simple en procédant comme suit :

1. Ouvrez la page Fonctions de la console Lambda.
2. Ouvrez le panneau d’aide en cliquant sur l’icône dans le coin supérieur droit de l’écran.



3. Sélectionnez Tutoriels.
4. Dans Créer une application Web simple, choisissez Démarrer le tutoriel.

API Gateway

Mise en route avec Lambda et API Gateway grâce aux ressources suivantes

- Suivez le tutoriel Utilisation de Lambda avec API Gateway pour créer une API REST intégrée à une fonction Lambda principale.
- Pour en savoir plus sur les différents types d’API proposés par API Gateway, consultez les sections suivantes du Guide du développeur Amazon API Gateway :
  - API Gateway REST APIs
  - API Gateway HTTP APIs
  - API Gateway WebSocket APIs
- Essayez un ou plusieurs des exemples de la section Tutoriels et ateliers du manuel Guide du développeur Amazon API Gateway.

Didacticiel : création d'un point de terminaison webhook à l'aide d'une URL de fonction Lambda

Dans ce didacticiel, vous créez une URL de fonction Lambda pour implémenter un point de terminaison webhook. Un webhook est une communication légère pilotée par les événements qui envoie automatiquement des données entre applications via HTTP. Vous pouvez utiliser un webhook pour recevoir des mises à jour immédiates sur les événements survenant dans un autre système, par exemple lorsqu'un nouveau client s'inscrit sur un site Web, qu'un paiement est traité ou qu'un fichier est chargé.

Avec Lambda, les webhooks peuvent être implémentés à l'aide de la fonction Lambda ou d'API Gateway. URLs URLs Les fonctions sont un bon choix pour les webhooks simples qui ne nécessitent pas de fonctionnalités telles que l'autorisation avancée ou la validation des demandes.

> **Tip**
>
> Si vous ne savez pas quelle est la solution la mieux adaptée à votre cas d'utilisation particulier, consultez the section called " URLs Function et Amazon API Gateway".

Conditions préalables

Pour terminer ce didacticiel, Python (version 3.8 ou ultérieure) ou Node.js (version 18 ou ultérieure) doit être installé sur votre machine locale.

Pour tester le point de terminaison à l'aide d'une requête HTTP, le didacticiel utilise curl, un outil de ligne de commande que vous pouvez utiliser pour transférer des données à l'aide de différents protocoles réseau. Reportez-vous à la documentation curl pour savoir comment installer l'outil si vous ne l'avez pas déjà.

Créer la fonction Lambda

Créez d'abord la fonction Lambda qui s'exécute lorsqu'une requête HTTP est envoyée à votre point de terminaison webhook. Dans cet exemple, l'application d'envoi envoie une mise à jour chaque fois qu'un paiement est soumis, et indique dans le corps de la requête HTTP si le paiement a été effectué avec succès. La fonction Lambda analyse la demande et agit en fonction du statut du paiement. Dans cet exemple, le code imprime simplement l'ID de commande pour le paiement, mais dans une application réelle, vous pouvez ajouter la commande à une base de données ou envoyer une notification.

La fonction implémente également la méthode d'authentification la plus couramment utilisée pour les webhooks, l'authentification des messages basée sur le hachage (HMAC). Avec cette méthode, les applications d'envoi et de réception partagent une clé secrète. L'application d'envoi utilise un algorithme de hachage pour générer une signature unique à l'aide de cette clé associée au contenu du message, et inclut la signature dans la demande de webhook sous forme d'en-tête HTTP. L'application réceptrice répète ensuite cette étape en générant la signature à l'aide de la clé secrète, et compare la valeur obtenue avec la signature envoyée dans l'en-tête de la demande. Si le résultat correspond, la demande est considérée comme légitime.

Créez la fonction à l'aide de la console Lambda avec l'environnement d'exécution Python ou Node.js.

Python

Créer la fonction Lambda

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Créez une fonction de type « Hello World » de base en procédant comme suit :
   a. Choisissez Créer une fonction.
   b. Sélectionnez Créer à partir de zéro.
   c. Sous Nom de la fonction, saisissez **`myLambdaWebhook`**.
   d. Pour Runtime, sélectionnez python3.14.
   e. Choisissez Créer une fonction.
3. Dans le volet Source du code, remplacez le code existant en copiant et en collant ce qui suit :

```
import json
import hmac
import hashlib
import os

def lambda_handler(event, context):

    # Get the webhook secret from environment variables
    webhook_secret = os.environ['WEBHOOK_SECRET']

    # Verify the webhook signature
    if not verify_signature(event, webhook_secret):
        return {
            'statusCode': 401,
            'body': json.dumps({'error': 'Invalid signature'})
```

```
        }

    try:
        # Parse the webhook payload
        payload = json.loads(event['body'])

        # Handle different event types
        event_type = payload.get('type')

        if event_type == 'payment.success':
            # Handle successful payment
            order_id = payload.get('orderId')
            print(f"Processing successful payment for order {order_id}")

            # Add your business logic here
            # For example, update database, send notifications, etc.

        elif event_type == 'payment.failed':
            # Handle failed payment
            order_id = payload.get('orderId')
            print(f"Processing failed payment for order {order_id}")

            # Add your business logic here

        else:
            print(f"Received unhandled event type: {event_type}")

        # Return success response
        return {
            'statusCode': 200,
            'body': json.dumps({'received': True})
        }

    except json.JSONDecodeError:
        return {
            'statusCode': 400,
            'body': json.dumps({'error': 'Invalid JSON payload'})
        }
    except Exception as e:
        print(f"Error processing webhook: {e}")
        return {
            'statusCode': 500,
            'body': json.dumps({'error': 'Internal server error'})
        }
```

```
def verify_signature(event, webhook_secret):
    """
    Verify the webhook signature using HMAC
    """
    try:
        # Get the signature from headers
        signature = event['headers'].get('x-webhook-signature')

        if not signature:
            print("Error: Missing webhook signature in headers")
            return False

        # Get the raw body (return an empty string if the body key doesn't
 exist)
        body = event.get('body', '')

        # Create HMAC using the secret key
        expected_signature = hmac.new(
            webhook_secret.encode('utf-8'),
            body.encode('utf-8'),
            hashlib.sha256
        ).hexdigest()

        # Compare the expected signature with the received signature to
 authenticate the message
        is_valid = hmac.compare_digest(signature, expected_signature)
        if not is_valid:
            print(f"Error: Invalid signature. Received: {signature}, Expected:
 {expected_signature}")
            return False

        return True
    except Exception as e:
        print(f"Error verifying signature: {e}")
        return False
```

4. Dans la section DÉPLOYER, choisissez Déployer pour mettre à jour le code de votre fonction.

Node.js

Créer la fonction Lambda

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Créez une fonction de type « Hello World » de base en procédant comme suit :
   a. Choisissez Créer une fonction.
   b. Sélectionnez Créer à partir de zéro.
   c. Sous Nom de la fonction, saisissez **`myLambdaWebhook`**.
   d. Pour Runtime, sélectionnez nodejs24.x.
   e. Choisissez Créer une fonction.
3. Dans le volet Source du code, remplacez le code existant en copiant et en collant ce qui suit :

```
import crypto from 'crypto';

export const handler = async (event, context) => {
    // Get the webhook secret from environment variables
    const webhookSecret = process.env.WEBHOOK_SECRET;

    // Verify the webhook signature
    if (!verifySignature(event, webhookSecret)) {
        return {
            statusCode: 401,
            body: JSON.stringify({ error: 'Invalid signature' })
        };
    }

    try {
        // Parse the webhook payload
        const payload = JSON.parse(event.body);

        // Handle different event types
        const eventType = payload.type;

        switch (eventType) {
            case 'payment.success': {
                // Handle successful payment
                const orderId = payload.orderId;
                console.log(`Processing successful payment for order
 ${orderId}`);
```

```
                // Add your business logic here
                // For example, update database, send notifications, etc.
                break;
            }

            case 'payment.failed': {
                // Handle failed payment
                const orderId = payload.orderId;
                console.log(`Processing failed payment for order ${orderId}`);

                // Add your business logic here
                break;
            }

            default:
                console.log(`Received unhandled event type: ${eventType}`);
        }

        // Return success response
        return {
            statusCode: 200,
            body: JSON.stringify({ received: true })
        };

    } catch (error) {
        if (error instanceof SyntaxError) {
            // Handle JSON parsing errors
            return {
                statusCode: 400,
                body: JSON.stringify({ error: 'Invalid JSON payload' })
            };
        }

        // Handle all other errors
        console.error('Error processing webhook:', error);
        return {
            statusCode: 500,
            body: JSON.stringify({ error: 'Internal server error' })
        };
    }
};

// Verify the webhook signature using HMAC
```

```
const verifySignature = (event, webhookSecret) => {
    try {
        // Get the signature from headers
        const signature = event.headers['x-webhook-signature'];

        if (!signature) {
            console.log('No signature found in headers:', event.headers);
            return false;
        }

        // Get the raw body (return an empty string if the body key doesn't
 exist)
        const body = event.body || '';

        // Create HMAC using the secret key
        const hmac = crypto.createHmac('sha256', webhookSecret);
        const expectedSignature = hmac.update(body).digest('hex');

        // Compare expected and received signatures
        const isValid = signature === expectedSignature;
        if (!isValid) {
            console.log(`Invalid signature. Received: ${signature}, Expected:
 ${expectedSignature}`);
            return false;
        }

        return true;
    } catch (error) {
        console.error('Error during signature verification:', error);
        return false;
    }
 };
```

4. Dans la section DÉPLOYER, choisissez Déployer pour mettre à jour le code de votre fonction.

Créer la clé de secret

Pour que la fonction Lambda authentifie la demande de webhook, elle utilise une clé secrète qu'elle partage avec l'application appelante. Dans cet exemple, la clé est stockée dans une variable d'environnement. Dans une application de production, n'incluez pas d'informations sensibles telles

que des mots de passe dans votre code de fonction. Créez plutôt un AWS Secrets Manager secret, puis utilisez l'extension Lambda AWS Parameters and Secrets pour récupérer vos informations d'identification dans votre fonction Lambda.

Créer et stocker la clé secrète du webhook

1. Générez une longue chaîne aléatoire à l'aide d'un générateur de nombres aléatoires sécurisé par cryptographie. Vous pouvez utiliser les extraits de code suivants en Python ou Node.js pour générer et imprimer un secret de 32 caractères, ou utiliser votre propre méthode préférée.

   Python

   Example code pour générer un secret

   ```
   import secrets
   webhook_secret = secrets.token_urlsafe(32)
   print(webhook_secret)
   ```

   Node.js

   Example code pour générer un secret (format du module ES)

   ```
   import crypto from 'crypto';
   let webhookSecret = crypto.randomBytes(32).toString('base64');
   console.log(webhookSecret)
   ```

2. Stockez la chaîne générée en tant que variable d'environnement pour votre fonction en procédant comme suit :

   a. Dans l'onglet Configuration de votre fonction, sélectionnez Variables d'environnement.

   b. Choisissez Modifier.

   c. Choisissez Ajouter une variable d'environnement.

   d. Pour Clé, saisissez **`WEBHOOK_SECRET`**, puis pour Valeur, saisissez le secret que vous avez généré à l'étape précédente.

   e. Choisissez Enregistrer.

Vous devrez réutiliser ce secret ultérieurement dans le didacticiel pour tester votre fonction. Veillez donc à le noter dès maintenant.

Créer le point de terminaison d'URL de fonction

Créez un point de terminaison pour votre webhook à l'aide d'une URL de fonction Lambda. Comme vous utilisez le type d'authentification `NONE` pour créer un point de terminaison avec accès public, toute personne disposant de l'URL peut invoquer votre fonction. Pour en savoir plus sur le contrôle de l'accès aux fonctions URLs, voirthe section called "Contrôle d'accès". Si vous avez besoin d'options d'authentification plus avancées pour votre webhook, pensez à utiliser API Gateway.

Créer le point de terminaison d'URL de fonction

1. Dans l'onglet Configuration de votre fonction, sélectionnez URL de la fonction.
2. Choisissez Create function URL (Créer une URL de fonction).
3. Pour Type d'authentification, sélectionnez NONE.
4. Choisissez Enregistrer.

Le point de terminaison de l'URL de fonction que vous venez de créer s'affiche dans le volet URL de fonction. Copiez le point de terminaison ; vous l'utiliserez ultérieurement dans le didacticiel.

Tester la fonction dans console

Avant d'utiliser une requête HTTP pour invoquer votre fonction à l'aide du point de terminaison d'URL, testez-la dans la console pour vérifier que votre code fonctionne comme prévu.

Pour vérifier la fonction dans la console, vous devez d'abord calculer une signature de webhook à l'aide du secret que vous avez généré plus tôt dans le didacticiel avec les données utiles JSON de test suivantes :

```
{
    "type": "payment.success",
    "orderId": "1234",
    "amount": "99.99"
}
```

Utilisez l'un des exemples de code Python ou Node.js suivants pour calculer la signature du webhook à l'aide de votre propre secret.

Python

Calculer la signature du webhook

1. Enregistrez le code suivant dans un fichier nommé `calculate_signature.py`. Remplacez le secret de webhook dans le code par votre propre valeur.

```
import secrets
import hmac
import json
import hashlib

webhook_secret = "arlbSDCP86n_1H90s0fL_Qb2NAHBIBQOyGI0X4Zay4M"

body = json.dumps({"type": "payment.success", "orderId": "1234", "amount":
 "99.99"})

signature = hmac.new(
            webhook_secret.encode('utf-8'),
            body.encode('utf-8'),
            hashlib.sha256
        ).hexdigest()

print(signature)
```

2. Calculez la signature en exécutant la commande suivante à partir du répertoire où vous avez enregistré le code. Copiez la signature émise par le code.

```
python calculate_signature.py
```

Node.js

Calculer la signature du webhook

1. Enregistrez le code suivant dans un fichier nommé `calculate_signature.mjs`. Remplacez le secret de webhook dans le code par votre propre valeur.

```
import crypto from 'crypto';

const webhookSecret = "arlbSDCP86n_1H90s0fL_Qb2NAHBIBQOyGI0X4Zay4M"
```

```
const body = "{\"type\": \"payment.success\", \"orderId\": \"1234\", \"amount\":
 \"99.99\"}";

let hmac = crypto.createHmac('sha256', webhookSecret);
let signature = hmac.update(body).digest('hex');

console.log(signature);
```

2. Calculez la signature en exécutant la commande suivante à partir du répertoire où vous avez enregistré le code. Copiez la signature émise par le code.

```
node calculate_signature.mjs
```

Vous pouvez désormais tester le code de votre fonction à l'aide d'une requête HTTP de test dans la console.

Tester la fonction dans console

1. Sélectionnez l'onglet Code correspondant à votre fonction.
2. Dans la section ÉVÉNEMENTS DE TEST, choisissez Créer un nouvel événement de test.
3. Dans Event Name (Nom de l'événement), saisissez **myEvent**.
4. Remplacez le JSON existant en copiant et en collant ce qui suit dans le volet JSON d'événement. Remplacez la signature de webhook par la valeur calculée à l'étape précédente.

```
{
  "headers": {
    "Content-Type": "application/json",
    "x-webhook-signature":
 "2d672e7a0423fab740fbc040e801d1241f2df32d2ffd8989617a599486553e2a"
  },
  "body": "{\"type\": \"payment.success\", \"orderId\": \"1234\", \"amount\":
 \"99.99\"}"
}
```

5. Choisissez Enregistrer.
6. Sélectionnez Invoquer .

   Vous devez voir des résultats similaires à ce qui suit :

Python

```
Status: Succeeded
Test Event Name: myEvent

Response:
{
  "statusCode": 200,
  "body": "{\"received\": true}"
}

Function Logs:
START RequestId: 50cc0788-d70e-453a-9a22-ceaa210e8ac6 Version: $LATEST
Processing successful payment for order 1234
END RequestId: 50cc0788-d70e-453a-9a22-ceaa210e8ac6
REPORT RequestId: 50cc0788-d70e-453a-9a22-ceaa210e8ac6 Duration: 1.55 ms Billed
 Duration: 2 ms Memory Size: 128 MB Max Memory Used: 36 MB Init Duration: 136.32
 ms
```

Node.js

```
Status: Succeeded
Test Event Name: myEvent

Response:
{
  "statusCode": 200,
  "body": "{\"received\":true}"
}

Function Logs:
START RequestId: e54fe6c7-1df9-4f05-a4c4-0f71cacd64f4 Version: $LATEST
2025-01-10T18:05:42.062Z e54fe6c7-1df9-4f05-a4c4-0f71cacd64f4 INFO Processing
 successful payment for order 1234
END RequestId: e54fe6c7-1df9-4f05-a4c4-0f71cacd64f4
REPORT RequestId: e54fe6c7-1df9-4f05-a4c4-0f71cacd64f4 Duration: 60.10 ms Billed
 Duration: 61 ms Memory Size: 128 MB Max Memory Used: 72 MB Init Duration:
 174.46 ms

Request ID: e54fe6c7-1df9-4f05-a4c4-0f71cacd64f4
```

Tester la fonction avec une requête HTTP

Utilisez l'outil de ligne de commande curl pour tester votre point de terminaison webhook.

Tester la fonction avec des requêtes HTTP

1. Dans un terminal ou un programme shell, exécutez la commande curl suivante. Remplacez l'URL par la valeur du point de terminaison de votre URL de fonction et remplacez la signature du webhook par la signature que vous avez calculée à l'aide de votre propre clé secrète.

```
curl -X POST https://ryqgmbx5xjzxahif6frvzikpre0bpvpf.lambda-url.us-west-2.on.aws/
 \
-H "Content-Type: application/json" \
-H "x-webhook-
signature: d5f52b76ffba65ff60ea73da67bdf1fc5825d4db56b5d3ffa0b64b7cb85ef48b" \
-d '{"type": "payment.success", "orderId": "1234", "amount": "99.99"}'
```

   Vous devriez voir la sortie suivante :

```
{"received": true}
```

2. Examinez les CloudWatch journaux de votre fonction pour confirmer qu'elle a correctement analysé la charge utile en procédant comme suit :

   a. Ouvrez la page du groupe Logs dans la CloudWatch console Amazon.

   b. Sélectionnez le groupe de journaux de votre fonction (`/aws/lambda/myLambdaWebhook`).

   c. Sélectionnez le flux de journaux le plus récent.

      Vous devez voir des résultats similaires à ce qui suit dans les journaux de votre fonction :

      Python

```
Processing successful payment for order 1234
```

      Node.js

```
2025-01-10T18:05:42.062Z e54fe6c7-1df9-4f05-a4c4-0f71cacd64f4 INFO
 Processing successful payment for order 1234
```

3. Confirmez que votre code détecte une signature non valide en exécutant la commande curl suivante. Remplacez l'URL par le point de terminaison de l'URL de votre propre fonction.

```
curl -X POST https://ryqgmbx5xjzxahif6frvzikpre0bpvpf.lambda-url.us-west-2.on.aws/
 \
-H "Content-Type: application/json" \
-H "x-webhook-signature: abcdefg" \
-d '{"type": "payment.success", "orderId": "1234", "amount": "99.99"}'
```

Vous devriez voir la sortie suivante :

```
{"error": "Invalid signature"}
```

Nettoyage de vos ressources

Vous pouvez maintenant supprimer les ressources que vous avez créées pour ce didacticiel, sauf si vous souhaitez les conserver. En supprimant AWS les ressources que vous n'utilisez plus, vous évitez des frais inutiles pour votre Compte AWS.

Pour supprimer la fonction Lambda

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction que vous avez créée.
3. Sélectionnez Actions, Supprimer.
4. Saisissez **`confirm`** dans la zone de saisie de texte et choisissez Delete (Supprimer).

Lorsque vous avez créé la fonction Lambda dans la console, Lambda a également créé un rôle d'exécution pour votre fonction.

Pour supprimer le rôle d'exécution

1. Ouvrez la page Roles (Rôles) de la console IAM.
2. Sélectionnez le rôle d'exécution que Lambda a créé. Le nom du rôle est au format `myLambdaWebhook-role-<random string>`.
3. Sélectionnez Delete (Supprimer).
4. Saisissez le nom du rôle dans le champ de saisie de texte et choisissez Delete (Supprimer).

Présentation de la mise à l'échelle de fonction Lambda

La simultanéité est le nombre de demandes en cours d'exécution que votre fonction AWS Lambda traite en même temps. Pour chaque demande simultanée, Lambda fournit une instance distincte de votre environnement d'exécution. Au fur et à mesure que vos fonctions reçoivent des demandes, Lambda gère automatiquement la mise à l'échelle du nombre d'environnements d'exécution jusqu'à ce que vous atteigniez la limite de simultanéité de votre compte. Par défaut, Lambda fournit à votre compte une limite de simultanéité totale de 1 000 exécutions simultanées pour toutes les fonctions d'une Région AWS. Pour répondre aux besoins spécifiques de votre compte, vous pouvez demander une augmentation du quota et configurer les contrôles de simultanéité au niveau des fonctions afin que vos fonctions critiques ne soient pas limitées.

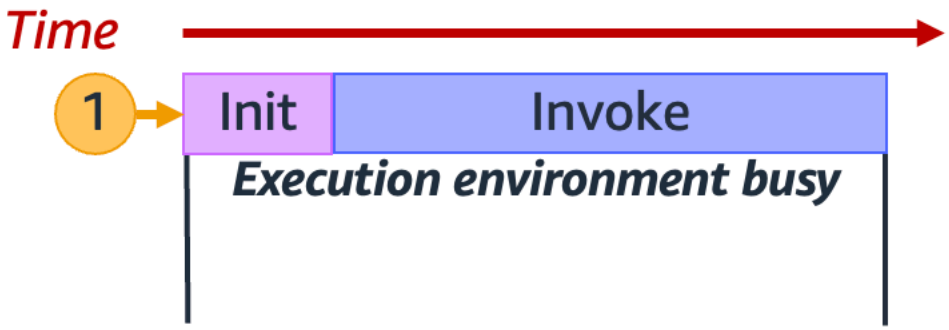
Cette rubrique explique la simultanéité et la mise à l'échelle des fonctions dans Lambda. À la fin de cette rubrique, vous serez en mesure de comprendre comment calculer la simultanéité, de visualiser les deux principales options de contrôle de la simultanéité (réservée et provisionnée), d'estimer les paramètres de contrôle de la simultanéité appropriés et de visualiser les métriques pour une optimisation supplémentaire.

Sections

- Comprendre et visualiser la simultanéité
- Calcul de la simultanéité d'une fonction
- Présentation de la simultanéité réservée et de la simultanéité allouée
- Présentation de la simultanéité et des requêtes par seconde
- Quotas de simultanéité
- Configuration de la simultanéité réservée pour une fonction
- Configuration de la simultanéité provisionnée pour une fonction
- Comportement de mise à l'échelle Lambda
- Surveillance de la simultanéité

Comprendre et visualiser la simultanéité

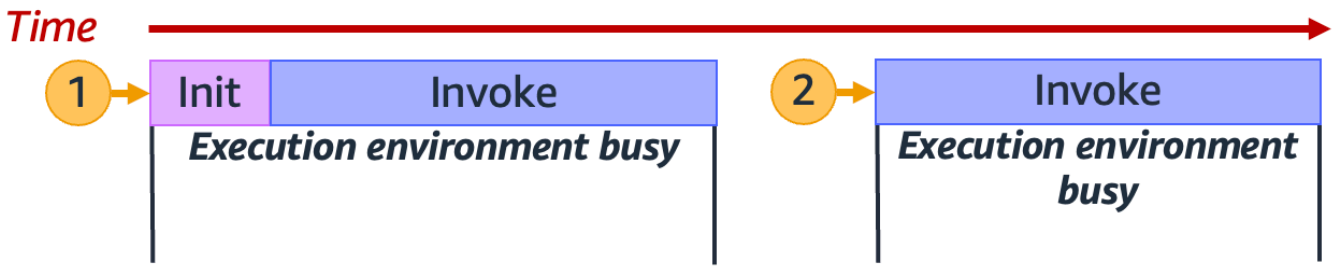
Lambda invoque votre fonction dans un environnement d'exécution sécurisé et isolé. Pour traiter une demande, Lambda doit d'abord initialiser un environnement d'exécution (la phase Init), avant de l'utiliser pour invoquer votre fonction (la phase Invoke) :



> **Note**
>
> Les durées réelles d'Init et Invoke peuvent varier en fonction de nombreux facteurs, tels que l'environnement d'exécution choisi et le code de la fonction Lambda. Le diagramme précédent n'est pas censé représenter les proportions exactes des durées des phases Init et Invoke.

Le diagramme précédent utilise un rectangle pour représenter un seul environnement d'exécution. Lorsque votre fonction reçoit sa toute première demande (représentée par le cercle jaune avec l'étiquette 1), Lambda crée un nouvel environnement d'exécution et exécute le code en dehors de votre gestionnaire principal pendant la phase Init. Ensuite, Lambda exécute le code du gestionnaire principal de votre fonction pendant la phase Invoke. Pendant tout ce processus, cet environnement d'exécution est occupé et ne peut pas traiter d'autres demandes.

Lorsque Lambda a fini de traiter la première demande, cet environnement d'exécution peut alors traiter des demandes supplémentaires pour la même fonction. Pour les demandes suivantes, Lambda n'a pas besoin de réinitialiser l'environnement.

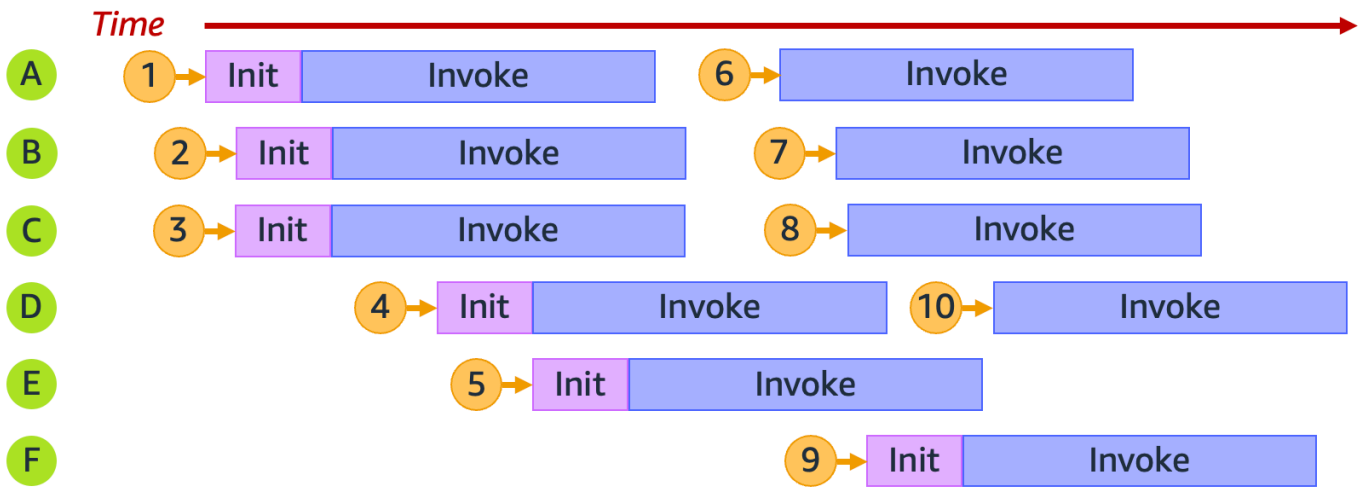


Dans le schéma précédent, Lambda réutilise l'environnement d'exécution pour traiter la deuxième demande (représentée par le cercle jaune avec l'étiquette 2).

Jusqu'à présent, nous nous sommes concentrés sur une seule instance de votre environnement d'exécution (c.-à-d. une simultanéité de 1). En pratique, Lambda peut avoir besoin de provisionner plusieurs instances d'environnement d'exécution en parallèle pour traiter toutes les demandes entrantes. Lorsque votre fonction reçoit une nouvelle demande, l'une des deux choses suivantes peut se produire :

- Si une instance d'environnement d'exécution pré-initialisée est disponible, Lambda l'utilise pour traiter la demande.
- Sinon, Lambda crée une nouvelle instance d'environnement d'exécution pour traiter la demande.

Par exemple, explorons ce qui se passe lorsque votre fonction reçoit 10 demandes :



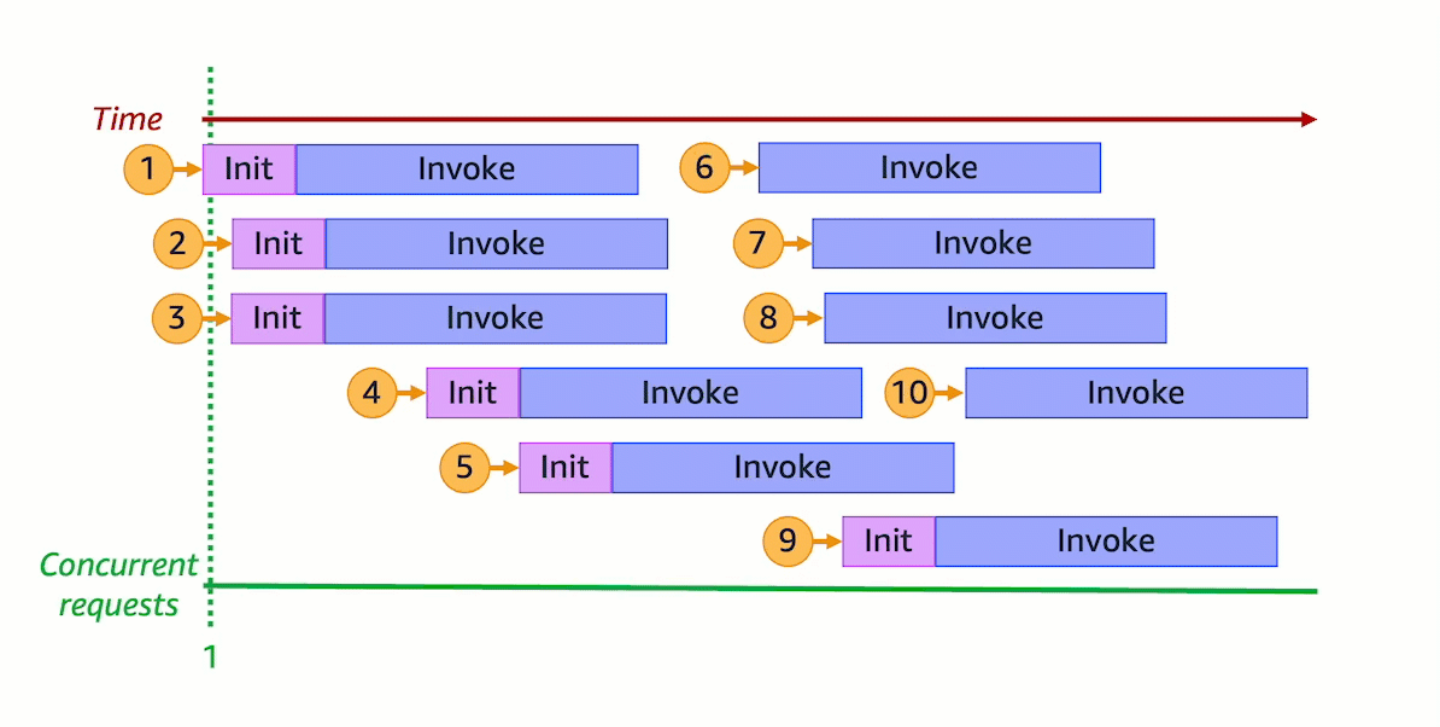
Dans le diagramme précédent, chaque plan horizontal représente une seule instance d'environnement d'exécution (étiquetée de A à F). Voici comment Lambda traite chaque demande :

Demande	Comportement de Lambda	Raisonnement
1	Alloue un nouvel environnement A	C'est la première demande ; aucune instance d'environnement d'exécution n'est disponible.

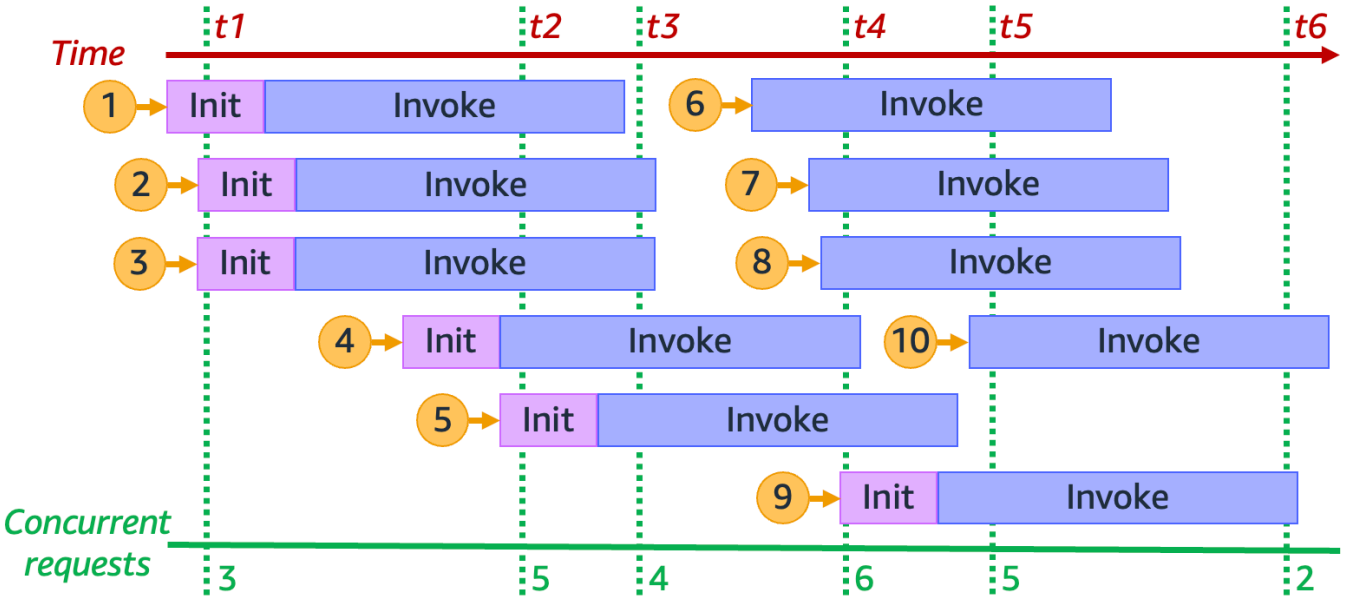
Demande	Comportement de Lambda	Raisonnement
2	Alloue un nouvel environnement B	L'instance A de l'environnement d'exécution existant est occupée.
3	Alloue un nouvel environnement C	Les instances d'environnement d'exécution existantes A et B sont toutes deux occupées.
4	Alloue un nouvel environnement D	Les instances d'environnement d'exécution existantes A, B et C sont toutes occupées.
5	Alloue un nouvel environnement E	Les instances A, B, C et D de l'environnement d'exécution existant sont toutes occupées.
6	Réutilise l'environnement A	L'instance d'environnement d'exécution A a fini de traiter la demande 1 et est maintenant disponible.
7	Réutilise l'environnement B	L'instance d'environnement d'exécution B a fini de traiter la demande 2 et est maintenant disponible.
8	Réutilise l'environnement C	L'instance d'environnement d'exécution C a terminé le traitement de la demande 3 et est maintenant disponible.

Demande	Comportement de Lambda	Raisonnement
9	Alloue un nouvel environnement F	Les instances d'environnement d'exécution existantes A, B, C, D et E sont toutes occupées.
10	Réutilise l'environnement D	L'instance d'environnement d'exécution D a fini de traiter la demande 4 et est maintenant disponible.

Au fur et à mesure que votre fonction reçoit des demandes simultanées, Lambda augmente le nombre d'instances d'environnement d'exécution en réponse. L'animation suivante suit le nombre de demandes simultanées dans le temps :



En figeant l'animation précédente à six points distincts dans le temps, nous obtenons le diagramme suivant :



Dans le diagramme précédent, nous pouvons tracer une ligne verticale à tout moment et compter le nombre d'environnements qui croisent cette ligne. Cela nous donne le nombre de demandes simultanées à ce moment précis. Par exemple, au moment `t1`, il y a trois environnements actifs servant trois demandes simultanées. Le nombre maximum de demandes simultanées dans cette simulation se produit au moment `t4`, lorsqu'il y a six environnements actifs servant six demandes simultanées.

En résumé, la simultanéité de votre fonction est le nombre de demandes simultanées qu'elle traite en même temps. En réponse à une augmentation de la simultanéité de votre fonction, Lambda fournit plus d'instances d'environnement d'exécution pour répondre à la demande.

Calcul de la simultanéité d'une fonction

En général, la simultanéité d'un système est la capacité de traiter plus d'une tâche simultanément. Dans Lambda, la simultanéité est le nombre de demandes en vol que votre fonction traite en même temps. Une façon rapide et pratique de mesurer la simultanéité d'une fonction Lambda est d'utiliser la formule suivante :

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

La simultanéité diffère des demandes par seconde. Par exemple, supposons que votre fonction reçoive 100 demandes par seconde en moyenne. Si la durée moyenne des demandes est de une seconde, il est vrai que la simultanéité est également de 100 :

```
Concurrency = (100 requests/second) * (1 second/request) = 100
```

Toutefois, si la durée moyenne des demandes est de 500 ms, la simultanéité est de 50 :

```
Concurrency = (100 requests/second) * (0.5 second/request) = 50
```

Que signifie une simultanéité de 50 dans la pratique ? Si la durée moyenne des demandes est de 500 ms, vous pouvez considérer qu'une instance de votre fonction est capable de traiter deux demandes par seconde. Ensuite, il faut 50 instances de votre fonction pour gérer une charge de 100 demandes par seconde. Une simultanéité de 50 signifie que Lambda doit fournir 50 instances d'environnement d'exécution pour gérer efficacement cette charge de travail sans être limité. Voici comment exprimer cela sous forme d'équation :

```
Concurrency = (100 requests/second) / (2 requests/second) = 50
```

Si votre fonction reçoit le double de demandes (200 demandes par seconde), mais ne nécessite que la moitié du temps pour traiter chaque demande (250 ms), la simultanéité est toujours de 50 :

```
Concurrency = (200 requests/second) * (0.25 second/request) = 50
```

Testez de votre compréhension de la simultanéité

Supposons que vous ayez une fonction qui prend, en moyenne, 200 ms pour s'exécuter. Pendant les pics de charge, vous observez 5 000 demandes par seconde. Quelle est la simultanéité de votre fonction pendant la charge de pointe ?

Réponse

La durée moyenne de la fonction est de 200 ms, soit 0,20 seconde. En utilisant la formule de simultanéité, vous pouvez entrer les chiffres pour obtenir une simultanéité de 1000 :

```
Concurrency = (5,000 requests/second) * (0.2 seconds/request) = 1,000
```

Autrement dit, une durée moyenne de fonction de 200 ms signifie que votre fonction peut traiter 5 demandes par seconde. Pour traiter la charge de travail de 5 000 demandes par seconde, vous avez besoin de 1 000 instances d'environnement d'exécution. La simultanéité est donc de 1 000 :

```
Concurrency = (5,000 requests/second) / (5 requests/second) = 1,000
```

Présentation de la simultanéité réservée et de la simultanéité allouée

Par défaut, votre compte dispose d'une limite de simultanéité de 1 000 exécutions simultanées pour toutes les fonctions d'une Région. Vos fonctions partagent ce groupe de 1 000 simultanéités sur une base à la demande. Vos fonctions sont limitées (c.-à-d. qu'elles commencent à abandonner des demandes) si vous n'avez plus de simultanéité disponible.

Certaines de vos fonctions peuvent être plus critiques que d'autres. Par conséquent, vous pouvez vouloir configurer les paramètres de simultanéité pour vous assurer que les fonctions critiques obtiennent la simultanéité dont elles ont besoin. Il existe deux types de contrôles de simultanéité : la simultanéité réservée et la simultanéité provisionnée.

- Utilisez la simultanéité réservée pour définir les nombres maximal et minimal d'instances simultanées afin de réserver une partie de la simultanéité de votre compte pour une fonction. Ceci est utile si vous ne voulez pas que d'autres fonctions prennent toute la simultanéité non réservée disponible. Lorsqu'une fonction dispose de la simultanéité réservée, aucune autre fonction ne peut utiliser cette simultanéité.
- Utilisez la simultanéité provisionnée pour pré-initialiser un certain nombre d'instances d'environnement pour une fonction. Ceci est utile pour réduire les latences de démarrage à froid.

Simultanéité réservée

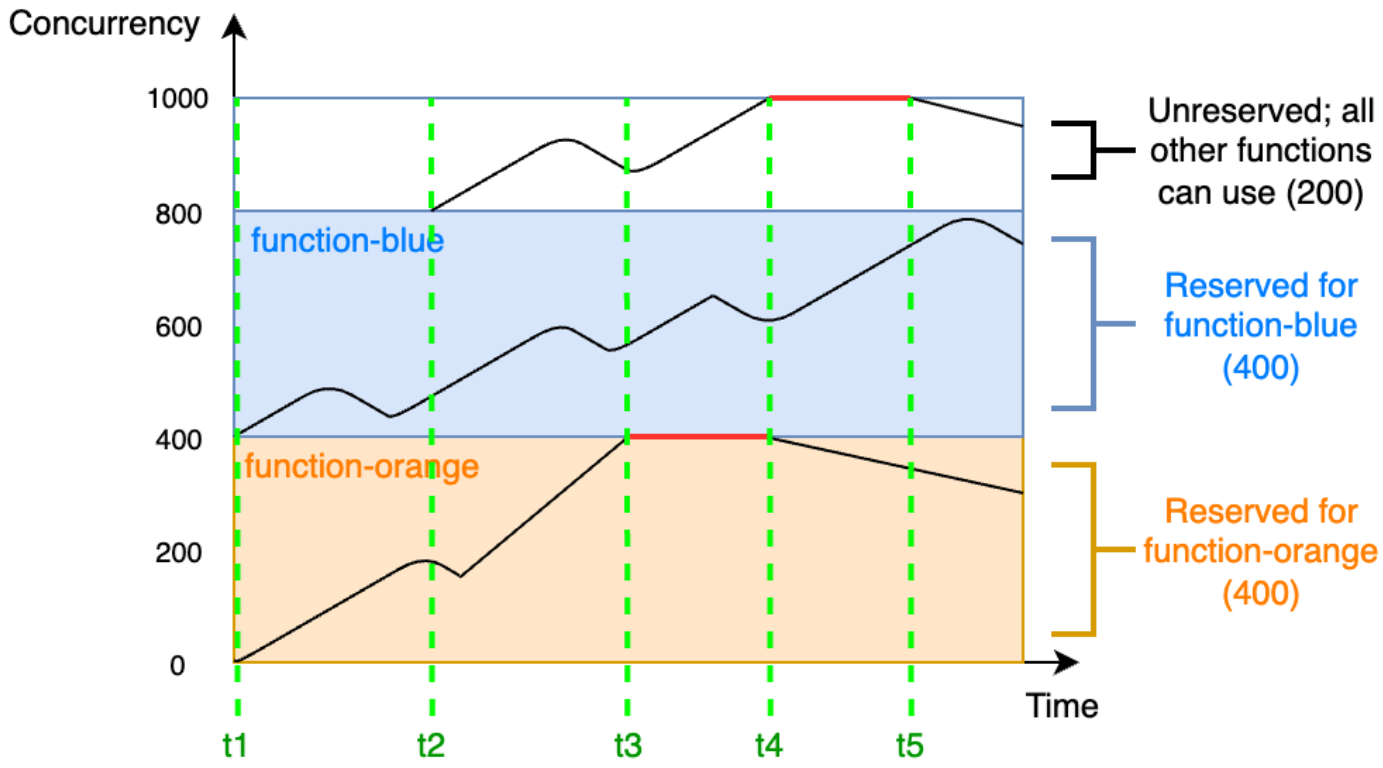
Si vous voulez garantir qu'une certaine quantité de simultanéité est disponible pour votre fonction à tout moment, utilisez la simultanéité réservée.

La simultanéité réservée définit les nombres maximum et minimum d'instances simultanées que vous voulez allouer à votre fonction. Lorsque vous consacrez une simultanéité réservée à une fonction, aucune autre fonction ne peut utiliser cette simultanéité. En d'autres termes, la définition de la simultanéité réservée peut avoir un impact sur le groupe de simultanéité disponible pour les

autres fonctions. Les fonctions qui n'ont pas de simultanéité réservée partagent le groupe restant de simultanéité non réservée.

La configuration de la simultanéité réservée compte dans la limite globale de simultanéité de votre compte. Il n'y a pas de frais pour la configuration de la simultanéité réservée pour une fonction.

Pour mieux comprendre la simultanéité réservée, considérez le diagramme suivant :



Dans ce diagramme, la limite de simultanéité de votre compte pour toutes les fonctions de cette Région est à la limite par défaut de 1 000. Supposons que vous ayez deux fonctions critiques, `function-blue` et `function-orange`, qui s'attendent régulièrement à recevoir des volumes d'invocations élevés. Vous décidez d'accorder 400 unités de simultanéité réservée à `function-blue`, et 400 unités de simultanéité réservée à `function-orange`. Dans cet exemple, toutes les autres fonctions de votre compte doivent se partager les 200 unités restantes de simultanéité non réservée.

Le diagramme présente cinq points d'intérêt :

- À `t1`, `function-orange` et `function-blue` commencent à recevoir des demandes. Chaque fonction commence à utiliser sa portion d'unités de simultanéité réservées.

- À `t2`, `function-orange` et `function-blue` reçoivent de plus en plus de demandes. Au même moment, vous déployez d'autres fonctions Lambda, qui commencent à recevoir des demandes. Vous n'allouez pas de simultanéité réservée à ces autres fonctions. Elles commencent à utiliser les 200 unités restantes de simultanéité non réservée.
- À `t3`, `function-orange` atteint la simultanéité maximale de 400. Bien qu'il existe une simultanéité non utilisée ailleurs dans le compte, `function-orange` ne peut pas y accéder. La ligne rouge indique que `function-orange` est limité et que Lambda peut abandonner des demandes.
- À `t4`, `function-orange` commence à recevoir moins de demandes et n'est plus limitée. Cependant, vos autres fonctions subissent un pic de trafic et commencent à être limitées. Bien qu'il y ait de la simultanéité inutilisée ailleurs dans votre compte, ces autres fonctions ne peuvent pas y accéder. La ligne rouge indique que vos autres fonctions sont limitées.
- À `t5`, les autres fonctions commencent à recevoir moins de demandes et ne sont plus limitées.

À partir de cet exemple, remarquez que la réservation de la simultanéité a les effets suivants :

- Votre fonction peut évoluer indépendamment des autres fonctions de votre compte. Toutes les fonctions de votre compte dans la même Région qui n'ont pas de simultanéité réservée partagent le groupe de simultanéité non réservée. Sans simultanéité réservée, d'autres fonctions peuvent utiliser toute votre simultanéité disponible. Cela empêche les fonctions critiques d'augmenter d'échelle si nécessaire.
- Votre fonction ne peut pas monter en puissance de façon incontrôlée. La simultanéité réservée limite la simultanéité maximale et minimale de votre fonction. Cela signifie que votre fonction ne peut pas utiliser la simultanéité réservée à d'autres fonctions, ou la simultanéité du groupe non réservé. En outre, la simultanéité réservée agit à la fois comme limite inférieure et supérieure : elle réserve la capacité spécifiée exclusivement à votre fonction tout en l'empêchant de dépasser cette limite. Vous pouvez réserver de la simultanéité pour empêcher votre fonction d'utiliser toute la simultanéité disponible dans votre compte, ou de surcharger les ressources en aval.
- Il se peut que vous ne puissiez pas utiliser toute la simultanéité disponible de votre compte. La réservation de la simultanéité compte dans la limite de simultanéité de votre compte, mais cela signifie également que les autres fonctions ne peuvent pas utiliser cette partie de la simultanéité réservée. Si votre fonction n'utilise pas toute la simultanéité que vous lui réservez, vous gaspillez effectivement cette simultanéité. Ce n'est pas un problème, sauf si d'autres fonctions de votre compte peuvent bénéficier de la simultanéité gaspillée.

Pour gérer les paramètres de simultanéité réservée pour vos fonctions, consultez Configuration de la simultanéité réservée pour une fonction.

Simultanéité provisionnée

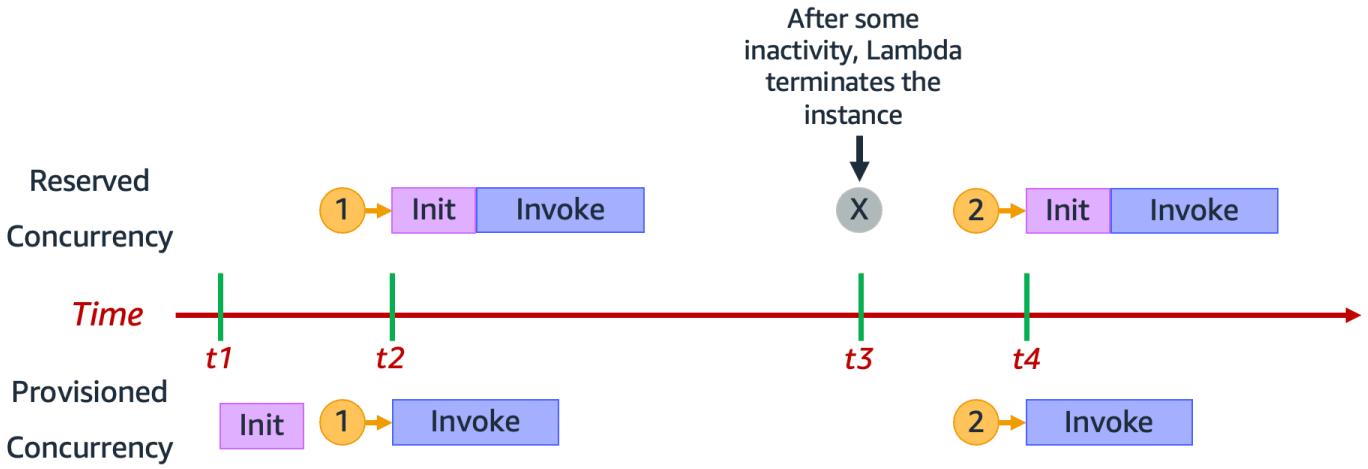
Vous utilisez la simultanéité réservée pour définir le nombre maximal d'environnements d'exécution réservés à une fonction Lambda. Toutefois, aucun de ces environnements n'est préinitialisé. Par conséquent, les invocations de votre fonction peuvent prendre plus de temps, car Lambda doit d'abord initialiser le nouvel environnement avant de pouvoir l'utiliser pour invoquer votre fonction. Lorsque Lambda doit initialiser un nouvel environnement pour effectuer une invocation , on parle de démarrage à froid. Pour atténuer les démarrages à froid, vous pouvez utiliser la simultanéité provisionnée.

La simultanéité provisionnée est le nombre d'environnements d'exécution pré-initialisés que vous voulez allouer à votre fonction. Si vous définissez la simultanéité provisionnée sur une fonction, Lambda initialise ce nombre d'environnements d'exécution afin qu'ils soient prêts à répondre immédiatement aux demandes de la fonction.

> **Note**
>
> L'utilisation de la simultanéité provisionnée entraîne des frais supplémentaires sur votre compte. Si vous travaillez avec les environnements d'exécution Java 11 ou Java 17, vous pouvez également utiliser Lambda SnapStart pour atténuer les problèmes de démarrage à froid sans frais supplémentaires. SnapStart utilise des instantanés mis en cache de votre environnement d'exécution pour améliorer de manière significative les performances de démarrage. Vous ne pouvez pas utiliser à la fois SnapStart et la simultanéité provisionnée sur la même version de fonction. Pour en savoir plus sur les fonctionnalités de SnapStart, les limitations et les régions prises en charge, consultez Améliorer les performances de démarrage avec Lambda SnapStart.

Lorsque vous utilisez la simultanéité provisionnée, Lambda recycle toujours les environnements d'exécution en arrière-plan. Par exemple, cela peut se produire après un échec d'invocation. Cependant, à tout moment, Lambda s'assure toujours que le nombre d'environnements pré-initialisés est égal à la valeur du paramètre de simultanéité provisionnée de votre fonction. Il est important de noter que même si vous utilisez la simultanéité provisionnée, vous pourriez toujours rencontrer un retard de démarrage à froid si Lambda doit réinitialiser l'environnement d'exécution.

En revanche, lorsque la simultanéité réservée est utilisée, Lambda peut résilier complètement un environnement après une période d'inactivité. Le diagramme suivant illustre cela en comparant le cycle de vie d'un environnement d'exécution unique lorsque vous configurez votre fonction en utilisant la simultanéité réservée par rapport à la simultanéité provisionnée.

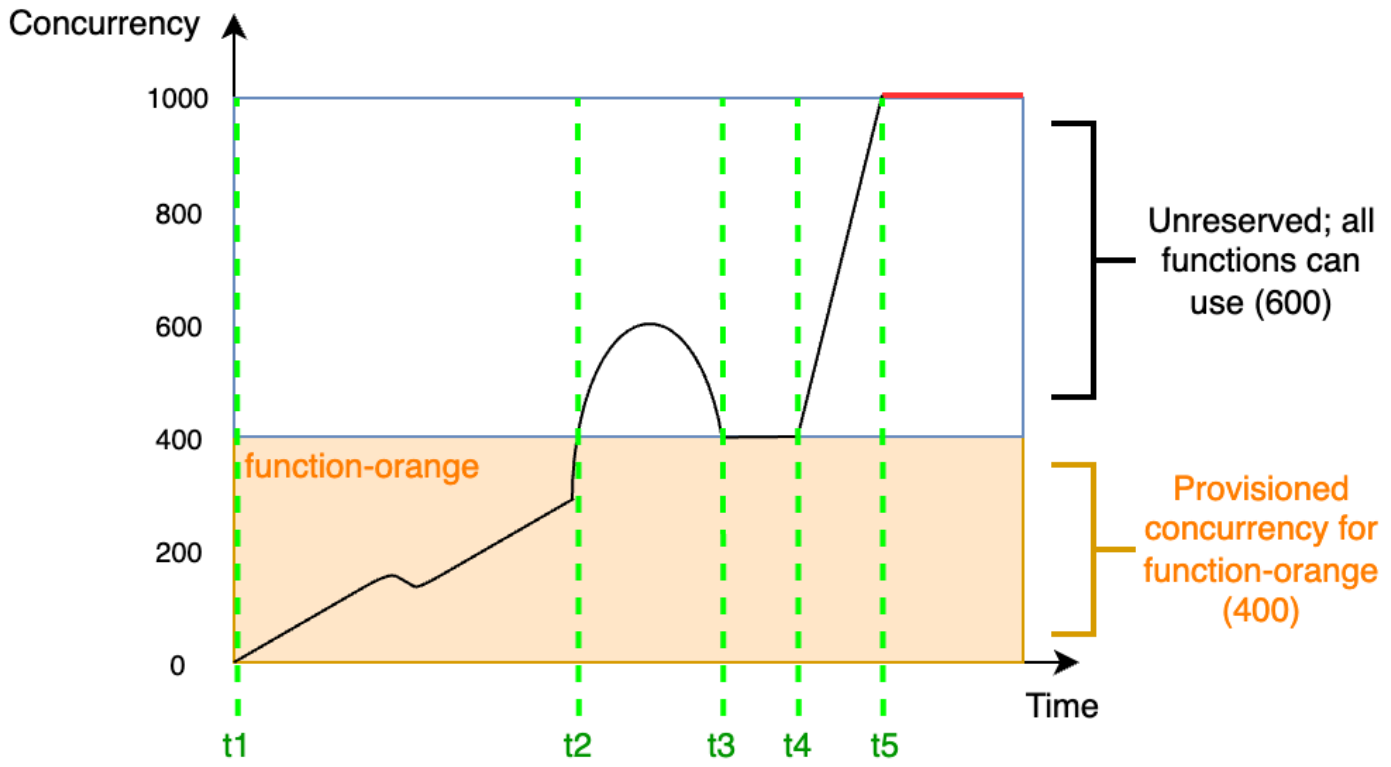


Le diagramme présente quatre points d'intérêt :

Heure	Simultanéité réservée	Simultanéité provisionnée
t1	Rien ne se passe.	Lambda pré-initialise une instance d'environnement d'exécution.
t2	La demande 1 arrive. Lambda doit initialiser une nouvelle instance d'environnement d'exécution.	La demande 1 arrive. Lambda utilise l'instance d'environ nement pré-initialisée.
t3	Après un certain temps d'inactivité, Lambda met fin à l'instance d'environnement active.	Rien ne se passe.
t4	La demande 2 arrive. Lambda doit initialiser une nouvelle	La demande 2 arrive. Lambda utilise l'instance d'environ nement pré-initialisée.

Heure	Simultanéité réservée	Simultanéité provisionnée
	instance d'environnement d'exécution.	

Pour mieux comprendre la simultanéité provisionnée, considérez le diagramme suivant :



Dans ce diagramme, vous avez une limite de simultanéité de compte de 1 000. Vous décidez d'accorder 400 unités de simultanéité provisionnée à `function-orange`. Toutes les fonctions de votre compte, y compris `function-orange`, peuvent utiliser les 600 unités restantes de simultanéité non réservée.

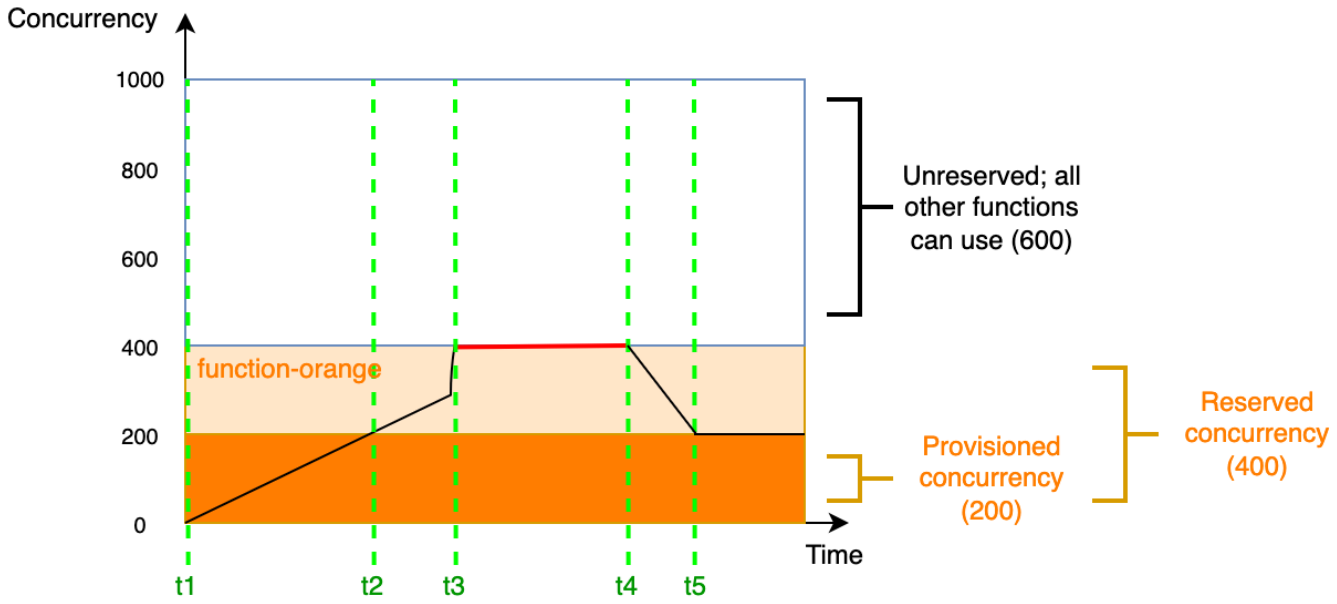
Le diagramme présente cinq points d'intérêt :

- À `t1`, `function-orange` commence à recevoir des demandes. Puisque Lambda a pré-initialisé 400 instances d'environnement d'exécution, `function-orange` est prête pour une invocation immédiat.
- À `t2`, `function-orange` atteint 400 demandes simultanées. Par conséquent, `function-orange` n'a plus de simultanéité provisionnée. Cependant, comme il reste de la simultanéité non

réservée, Lambda peut l'utiliser pour traiter des demandes supplémentaires vers `function-orange` (il n'y a pas de limitation). Lambda doit créer de nouvelles instances pour répondre à ces demandes, et votre fonction peut subir des latences de démarrage à froid.

- À `t3`, `function-orange` revient à 400 demandes simultanées après un bref pic de trafic. Lambda est à nouveau capable de traiter toutes les demandes sans latence de démarrage à froid.
- À `t4`, les fonctions de votre compte subissent un pic de trafic. Cette rafale peut provenir de `function-orange` ou de toute autre fonction de votre compte. Lambda utilise la simultanéité sans réserve pour traiter ces demandes.
- À `t5`, les fonctions de votre compte atteignent la limite maximale de simultanéité de 1 000 et sont limitées.

L'exemple précédent ne prenait en compte que la simultanéité provisionnée. En pratique, vous pouvez définir à la fois la simultanéité provisionnée et la simultanéité réservée sur une fonction. Vous pourriez le faire si vous aviez une fonction qui gère une charge constante d'invocations le week-end, mais qui connaît régulièrement des pics de trafic le week-end. Dans ce cas, vous pourriez utiliser la simultanéité provisionnée pour définir une quantité de base d'environnements pour traiter les demandes pendant les jours de la semaine, et utiliser la simultanéité réservée pour gérer les pics de trafic du week-end. Considérez le diagramme suivant :



Dans ce diagramme, supposons que vous configurez 200 unités de simultanéité provisionnée et 400 unités de simultanéité réservée pour `function-orange`. Parce que vous avez configuré la

simultanéité réservée, `function-orange` ne peut utiliser aucune des 600 unités de simultanéité non réservée.

Ce diagramme comporte cinq points d'intérêt :

- À `t1`, `function-orange` commence à recevoir des demandes. Puisque Lambda a pré-initialisé 200 instances d'environnement d'exécution, `function-orange` est prête pour une invocation immédiat.
- À `t2`, `function-orange` utilise toute sa simultanéité provisionnée. `function-orange` peut continuer à servir les demandes en utilisant la simultanéité réservée, mais ces demandes peuvent subir des latences de démarrage à froid.
- À `t3`, `function-orange` atteint 400 demandes simultanées. Par conséquent, `function-orange` utilise toute sa simultanéité réservée. Comme `function-orange` ne peut pas utiliser la simultanéité non réservée, les demandes commencent à être limitées.
- À `t4`, `function-orange` commence à recevoir moins de demandes et ne se limite plus.
- À `t5`, `function-orange` tombe à 200 demandes simultanées, de sorte que toutes les demandes peuvent à nouveau utiliser la simultanéité provisionnée (c.-à-d. sans latence de démarrage à froid).

La simultanéité réservée et la simultanéité provisionnée comptent toutes deux dans la limite de simultanéité de votre compte et dans les quotas régionaux. En d'autres termes, l'allocation de la simultanéité réservée et provisionnée peut avoir un impact sur le groupe de simultanéité disponible pour les autres fonctions. La configuration de la simultanéité provisionnée entraîne des frais sur votre Compte AWS.

> **Note**
>
> Si la simultanéité provisionnée sur les versions et alias d'une fonction s'ajoute à la simultanéité réservée de la fonction, toutes les invocations s'exécutent sur la simultanéité provisionnée. Cette configuration a également pour effet de limiter la version non publiée de la fonction (`$LATEST`), ce qui empêche son exécution. Vous ne pouvez pas allouer plus de simultanéité provisionnée que de simultanéité réservée pour une fonction.

Pour gérer les paramètres de simultanéité provisionnée pour vos fonctions, consultez Configuration de la simultanéité provisionnée pour une fonction. Pour automatiser la mise à l'échelle de la concurrence provisionnée en fonction d'une planification ou de l'utilisation de l'application, consultez Utilisation d'Application Auto Scaling pour automatiser la gestion de la simultanéité provisionnée.

Comment Lambda alloue la simultanéité provisionnée

La simultanéité provisionnée n'est pas mise en ligne immédiatement après la configuration. Lambda commence à allouer la simultanéité approvisionnée après une phase de préparation d'une ou deux minutes. Pour chaque fonction, Lambda peut allouer jusqu'à 6 000 environnements d'exécution par minute, quel que soit l'Région AWS. C'est exactement le même que le taux de mise à l'échelle de la simultanéité pour les fonctions.

Lorsque vous soumettez une demande d'allocation de simultanéité provisionnée, vous ne pouvez accéder à aucun de ces environnements tant que Lambda n'a pas terminé de les allouer. Par exemple, si vous demandez 5 000 simultanéités allouées, aucune de vos requêtes ne peut utiliser la simultanéité allouée avant que Lambda termine l'allocation des 5 000 environnements d'exécution.

Comparaison de la simultanéité réservée et de la simultanéité provisionnée

Le tableau suivant résume et compare la simultanéité réservée et provisionnée.

Rubrique	Simultanéité réservée	Simultanéité provisionnée
Définition	Nombre maximal d'instances d'environnement d'exécution pour votre fonction.	Nombre défini d'instances d'environnement d'exécution pré-provisionnées pour votre fonction.
Comportement de provision nement	Lambda provisionne de nouvelles instances sur une base à la demande.	Lambda pré-provisionne les instances (c.-à-d. avant que votre fonction ne commence à recevoir des demandes).
Comportement de démarrage à froid	Latence de démarrage à froid possible, puisque Lambda doit créer de nouvelles instances à la demande.	Latence de démarrage à froid impossible, puisque Lambda ne doit pas créer d'instances à la demande.
Comportement de limitation	La fonction est limitée lorsque la limite de simultanéité réservée est atteinte.	Si la simultanéité réservée n'est pas définie : la fonction utilise la simultanéité non réservée lorsque la limite de

Rubrique	Simultanéité réservée	Simultanéité provisionnée
		simultanéité provisionnée est atteinte. Si la simultanéité réservée est définie : la fonction est limitée lorsque la limite de simultané ité réservée est atteinte.
Comportement par défaut si non défini	La fonction utilise la simultané ité non réservée disponible dans votre compte.	Lambda ne pré-provisionne pas d'instances. Au lieu de cela, si la simultanéité réservée n'est pas définie : la fonction utilise la simultanéité non réservée disponible dans votre compte. Si la simultanéité réservée est définie : la fonction utilise la simultanéité réservée.
Tarification	Pas de frais supplémentaires.	Entraîne des frais supplémen taires.

Présentation de la simultanéité et des requêtes par seconde

Comme indiqué dans la section précédente, la simultanéité diffère des demandes par seconde. Cette distinction est particulièrement importante lorsque l'on travaille avec des fonctions dont la durée moyenne des requêtes est inférieure à 100 ms.

Pour toutes les fonctions de votre compte, Lambda applique une limite de requêtes par seconde égale à 10 fois le nombre de requêtes simultanées de votre compte. Par exemple, étant donné que la limite de simultanéité par défaut est de 1 000, les fonctions de votre compte peuvent traiter un maximum de 10 000 requêtes par seconde.

Prenons l'exemple d'une fonction dont la durée moyenne des demandes est de 50 ms. À raison de 20 000 requêtes par seconde, voici la simultanéité de cette fonction :

```
Concurrency = (20,000 requests/second) * (0.05 second/request) = 1,000
```

Sur la base de ce résultat, vous pouvez vous attendre à ce que la limite de simultanéité de compte de 1 000 soit suffisante pour gérer cette charge. Toutefois, en raison de la limite de 10 000 requêtes par seconde, votre fonction ne peut traiter que 10 000 requêtes par seconde sur un total de 20 000 requêtes. Cette fonction est soumise à une limitation.

La leçon à tirer est que vous devez tenir compte à la fois de la simultanéité et des demandes par seconde lorsque vous configurez les paramètres de simultanéité pour vos fonctions. Dans ce cas, vous devez demander une augmentation de la limite de simultanéité des comptes à 2 000, car cela porterait le nombre total de requêtes par seconde à 20 000.

> **Note**
>
> Sur la base de cette limite de requêtes par seconde, il est faux de dire que chaque environnement d'exécution Lambda ne peut traiter qu'un maximum de 10 requêtes par seconde. Au lieu d'observer la charge sur un environnement d'exécution individuel, Lambda ne prend en compte que la simultanéité globale et le nombre total de requêtes par seconde lors du calcul de vos quotas.

Testez votre compréhension de la concurrence (fonctions inférieures à 100 ms)

Supposons que vous ayez une fonction qui prend, en moyenne, 20 ms pour s'exécuter. Pendant les pics de charge, vous observez 30 000 requêtes par seconde. Quelle est la simultanéité de votre fonction pendant la charge de pointe ?

Réponse

La durée moyenne de la fonction est de 20 ms, soit 0,02 seconde. En utilisant la formule de simultanéité, vous pouvez entrer les chiffres pour obtenir une simultanéité de 600 :

```
Concurrency = (30,000 requests/second) * (0.02 seconds/request) = 600
```

Par défaut, la limite de simultanéité de compte de 1 000 semble suffisante pour gérer cette charge. Cependant, la limite de 10 000 requêtes par seconde n'est pas suffisante pour traiter les 30 000 requêtes entrantes par seconde. Pour répondre pleinement aux 30 000 requêtes, vous devez demander une augmentation de la limite de simultanéité des comptes à 3 000 ou plus.

La limite de requêtes par seconde s'applique à tous les quotas de Lambda impliquant une simultanéité. En d'autres termes, cela s'applique aux fonctions synchrones à la demande, aux fonctions qui utilisent la simultanéité allouée et au comportement de mise à l'échelle de simultanéité. Par exemple, voici quelques scénarios dans lesquels vous devez examiner attentivement vos limites de simultanéité et de requêtes par seconde :

- Une fonction utilisant la simultanéité à la requête peut connaître une augmentation de 500 requêtes simultanées toutes les 10 secondes, ou de 5 000 requêtes par seconde toutes les 10 secondes, selon la première éventualité.
- Supposons que vous disposiez d'une fonction dotée d'une allocation de simultanéité allouée de 10. Cette fonction se transforme en simultanéité à la requête après 10 requêtes simultanées ou 100 requêtes par seconde, selon la première éventualité.

Quotas de simultanéité

Lambda définit des quotas pour la quantité totale de simultanéité que vous pouvez utiliser sur toutes les fonctions d'une Région. Ces quotas existent à deux niveaux :

- Au niveau du compte, vos fonctions peuvent avoir jusqu'à 1 000 unités de simultanéité par défaut. Pour augmenter cette limite, consultez Demande d'augmentation de quota dans le Guide de l'utilisateur Service Quotas.
- Au niveau de la fonction, vous pouvez réserver par défaut jusqu'à 900 unités de simultanéité pour l'ensemble de vos fonctions. Quelle que soit la limite totale de simultanéité de votre compte, Lambda réserve toujours 100 unités de simultanéité à vos fonctions qui ne réservent pas explicitement la simultanéité. Par exemple, si vous avez augmenté la limite de simultanéité de votre compte à 2 000, vous pouvez réserver jusqu'à 1 900 unités de simultanéité au niveau de la fonction.
- Au niveau du compte comme au niveau de la fonction, Lambda impose également une limite de requêtes par seconde égale à 10 fois le quota de simultanéité correspondant. Cela s'applique par exemple à la simultanéité au niveau du compte, aux fonctions utilisant la simultanéité à la demande, aux fonctions utilisant la simultanéité allouée et au comportement de mise à l'échelle de la simultanéité. Pour de plus amples informations, consultez the section called "Présentation de la simultanéité et des requêtes par seconde".

Pour vérifier le quota de simultanéité au niveau de votre compte actuel, utilisez AWS Command Line Interface (AWS CLI) pour exécuter la commande suivante :

```
aws lambda get-account-settings
```

Vous devriez voir une sortie semblable à la suivante :

```
{
    "AccountLimit": {
        "TotalCodeSize": 80530636800,
        "CodeSizeUnzipped": 262144000,
        "CodeSizeZipped": 52428800,
        "ConcurrentExecutions": 1000,
        "UnreservedConcurrentExecutions": 900
    },
    "AccountUsage": {
        "TotalCodeSize": 410759889,
        "FunctionCount": 8
    }
}
```

`ConcurrentExecutions` est le quota total de simultanéité au niveau de votre compte. `UnreservedConcurrentExecutions` est la quantité de concurrence réservée que vous pouvez encore allouer à vos fonctions.

Au fur et à mesure que votre fonction reçoit des demandes, Lambda augmente automatiquement le nombre d'environnements d'exécution pour traiter ces demandes jusqu'à ce que votre compte atteigne sa limite de simultanéité. Toutefois, pour éviter une mise à l'échelle excessive en réponse à des pics de trafic soudains, Lambda limite la rapidité avec laquelle vos fonctions peuvent être mises à l'échelle. Ce taux de mise à l'échelle de la simultanéité est la vitesse maximale à laquelle les fonctions de votre compte peuvent se mettre à l'échelle en réponse à l'augmentation du nombre de requêtes. (C'est-à-dire la rapidité avec laquelle Lambda peut créer de nouveaux environnements d'exécution.) Le aux de mise à l'échelle de la simultanéité est différent de la limite de simultanéité au niveau du compte, qui est le montant total de simultanéité disponible pour vos fonctions.

Dans chaque Région AWS et pour chaque fonction, votre taux de mise à l'échelle de la simultanéité est de 1 000 instances d'environnement d'exécution toutes les 10 secondes (ou 10 000 requêtes par seconde toutes les 10 secondes). En d'autres termes, toutes les 10 secondes, Lambda peut allouer au maximum 1 000 instances d'environnement d'exécution supplémentaires, ou accueillir 10 000 requêtes supplémentaires par seconde, pour chacune de vos fonctions.

En général, il n'est pas nécessaire de se soucier de cette limitation. La vitesse de mise à l'échelle de Lambda est suffisante dans la plupart des cas d'utilisation.

Il est important de noter que le taux de mise à l'échelle de la simultanéité est une limite au niveau de la fonction. Cela signifie que chaque fonction de votre compte peut être mise à l'échelle indépendamment des autres fonctions.

Pour plus d'informations sur les comportements de mise à l'échelle, consultez Comportement de mise à l'échelle Lambda.

Configuration de la simultanéité réservée pour une fonction

Dans Lambda, la simultanéité est le nombre de demandes en vol que votre fonction traite en même temps. Il existe deux types de contrôles de la simultanéité disponibles :

- Simultanéité réservée : définit les nombres maximum et minimum d'instances simultanées allouées à votre fonction. Lorsqu'une fonction dispose de la simultanéité réservée, aucune autre fonction ne peut utiliser cette simultanéité. La simultanéité réservée est utile pour garantir que vos fonctions les plus critiques disposent toujours d'une simultanéité suffisante pour traiter les requêtes entrantes. En outre, la simultanéité réservée peut être utilisée pour limiter la simultanéité afin d'éviter de surcharger les ressources en aval, comme les connexions aux bases de données. La simultanéité réservée agit à la fois comme limite inférieure et supérieure : elle réserve la capacité spécifiée exclusivement à votre fonction tout en l'empêchant de dépasser cette limite. Il n'y a pas de frais supplémentaires pour la configuration de la simultanéité réservée pour une fonction.
- La simultanéité provisionnée est le nombre d'environnements d'exécution pré-initialisés alloués à votre fonction. Ces environnements d'exécution sont prêts à répondre immédiatement aux demandes de fonctions entrantes. La simultanéité allouée est utile pour réduire les latences de démarrage à froid pour les fonctions et est conçue pour rendre les fonctions disponibles avec des temps de réponse de l'ordre de quelques dixaines de millisecondes. En général, ce sont les charges de travail interactives qui tirent le meilleur parti de cette fonctionnalité. Il s'agit des applications dont les utilisateurs lancent des requêtes, telles que les applications Web et mobiles, et qui sont les plus sensibles à la latence. Les charges de travail asynchrones, telles que les pipelines de traitement des données, sont souvent moins sensibles à la latence et ne nécessitent donc généralement pas de simultanéité allouée. La configuration de la simultanéité provisionnée entraîne des frais supplémentaires sur votre Compte AWS.

Cette rubrique explique comment gérer et configurer la simultanéité réservée. Pour une présentation conceptuelle de ces deux types de contrôles de simultanéité, consultez Simultanéité réservée et simultanéité provisionnée. Pour plus d'informations sur la configuration de la simultanéité provisionnée, consultez the section called "Configuration de la simultanéité provisionnée".

> **Note**
>
> Les fonctions Lambda liées à un mappage des sources d'événements Amazon MQ ont une simultanéité maximale par défaut. Pour Apache Active MQ, le nombre maximum d'instances simultanées est de 5. Pour Rabbit MQ, le nombre maximum d'instances simultanées est de 1. La définition d'une simultanéité réservée ou provisionnée pour votre fonction ne modifie pas

ces limites. Pour demander une augmentation de la simultanéité maximale par défaut lors de l'utilisation d'Amazon MQ, contactez Support.

Sections

- Configuration de la simultanéité réservée
- Estimation précise de la simultanéité réservée requise pour une fonction

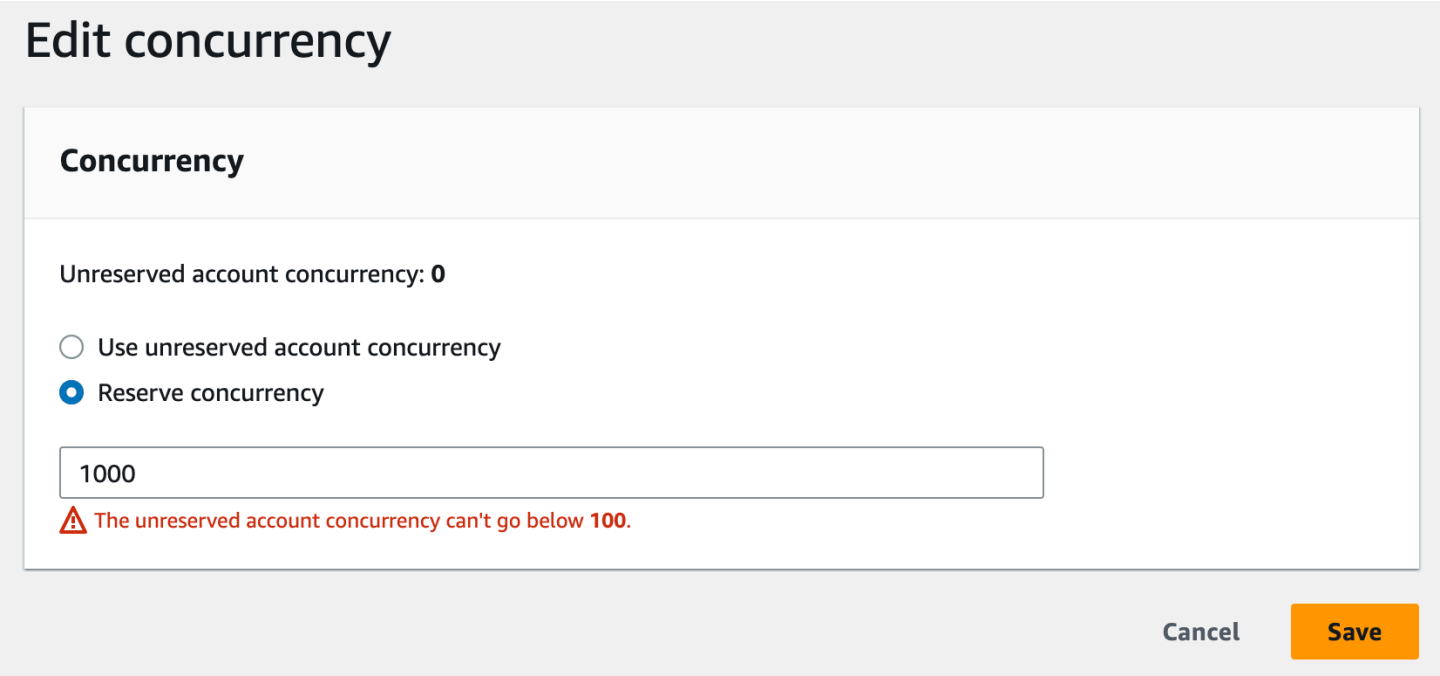
Configuration de la simultanéité réservée

Vous pouvez configurer les paramètres de simultanéité réservés pour une fonction à l'aide de la console Lambda ou de l'API Lambda.

Réserver la simultanéité pour une fonction (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction pour laquelle vous souhaitez réserver la simultanéité.
3. Sélectionnez Configuration, puis Concurrency (Simultanéité).
4. Sous Concurrency (Simultanéité), choisissez Edit (Modifier).
5. Choisissez Reserve concurrency (Réserver la simultanéité). Saisissez la quantité de simultanéité à réserver pour la fonction.
6. Choisissez Enregistrer.

Vous pouvez réserver jusqu'à la valeur de la simultanéité du compte non réservé moins 100. Les 100 unités de simultanéité restantes concernent les fonctions qui n'utilisent pas la simultanéité réservée. Par exemple, si votre compte a une limite de concurrence de 1 000, vous ne pouvez pas réserver les 1 000 unités de concurrence pour une seule fonction.



Le fait de réserver la simultanéité à une fonction peut avoir des conséquences sur le groupe de simultanéité disponible pour d'autres fonctions. Par exemple, si vous réservez 100 unités de simultanéité pour `function-a`, les autres fonctions de votre compte doivent partager les 900 unités de simultanéité restantes, même si `function-a` n'utilisent pas les 100 unités de simultanéité réservées.

Pour limiter intentionnellement une fonction, définissez la simultanéité réservée à zéro. Cela empêche votre fonction de traiter des événements jusqu'à ce que vous supprimiez la limite.

Pour configurer la simultanéité réservée avec l'API Lambda, utilisez les opérations d'API suivantes.

- PutFunctionConcurrency
- GetFunctionConcurrency
- DeleteFunctionConcurrency

Par exemple, pour configurer la simultanéité réservée avec la AWS Command Line Interface (CLI), utilisez la commande `put-function-concurrency`. La commande suivante réserve 100 unités de simultanéité pour une fonction nommée `my-function` :

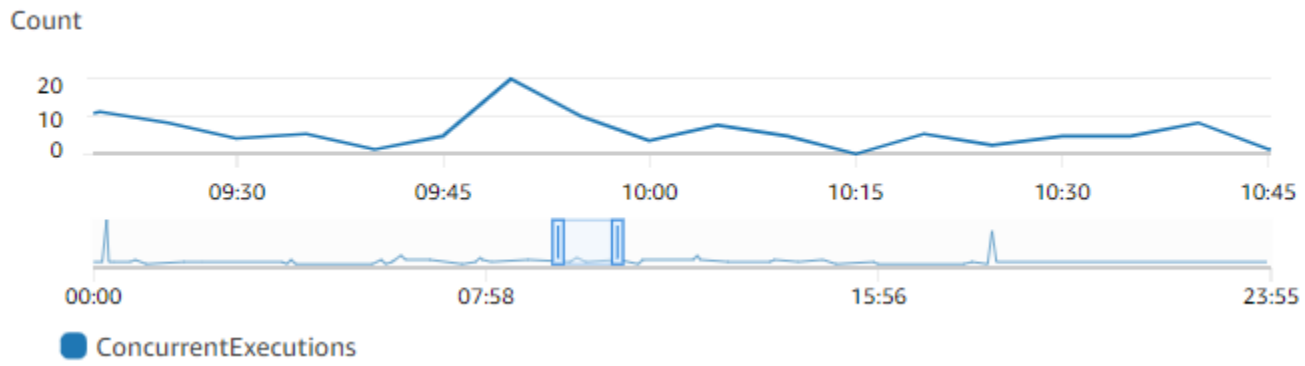
```
aws lambda put-function-concurrency --function-name my-function \
    --reserved-concurrent-executions 100
```

Vous devriez voir une sortie semblable à la suivante :

```
{
    "ReservedConcurrentExecutions": 100
}
```

Estimation précise de la simultanéité réservée requise pour une fonction

Si votre fonction sert actuellement du trafic, vous pouvez facilement visualiser ses métriques de simultanéité à l'aide des métriques CloudWatch. Plus précisément, la métrique `ConcurrentExecutions` vous montre le nombre d'invocations simultanées pour chaque fonction de votre compte.



Le graphique précédent indique que cette fonction répond à une moyenne de 5 à 10 demandes simultanées à tout moment, et qu'elle atteint un pic de 20 demandes lors d'une journée typique. Supposons qu'il y ait beaucoup d'autres fonctions dans votre compte. Si cette fonction est essentielle à votre application et que vous ne voulez pas perdre de demandes, utilisez un nombre supérieur ou égal à 20 comme paramètre de simultanéité réservé.

Par ailleurs, rappelez-vous que vous pouvez également calculer la simultanéité à l'aide de la formule suivante :

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

En multipliant le nombre moyen de demandes par seconde par la durée moyenne des demandes en secondes, vous obtenez une estimation approximative du niveau de simultanéité que vous devez réserver. Vous pouvez estimer les demandes moyennes par seconde à l'aide de la métrique `Invocation`, et la durée moyenne des demandes en secondes à l'aide de la métrique `Duration`. Pour plus d'informations, consultez Utilisation des métriques CloudWatch avec Lambda.

Familiarisez-vous avec vos contraintes de débit en amont et en aval. Bien que les fonctions Lambda se mettent à l'échelle parfaitement, les dépendances en amont et en aval peuvent avoir des mêmes capacités de débit différentes. Si vous devez limiter le niveau de mise à l'échelle de votre fonction, configurez la simultanéité réservée sur votre fonction.

Configuration de la simultanéité provisionnée pour une fonction

Dans Lambda, la simultanéité est le nombre de demandes en vol que votre fonction traite en même temps. Il existe deux types de contrôles de la simultanéité disponibles :

- Simultanéité réservée : définit les nombres maximum et minimum d'instances simultanées allouées à votre fonction. Lorsqu'une fonction dispose de la simultanéité réservée, aucune autre fonction ne peut utiliser cette simultanéité. La simultanéité réservée est utile pour garantir que vos fonctions les plus critiques disposent toujours d'une simultanéité suffisante pour traiter les requêtes entrantes. En outre, la simultanéité réservée peut être utilisée pour limiter la simultanéité afin d'éviter de surcharger les ressources en aval, comme les connexions aux bases de données. La simultanéité réservée agit à la fois comme limite inférieure et supérieure : elle réserve la capacité spécifiée exclusivement à votre fonction tout en l'empêchant de dépasser cette limite. Il n'y a pas de frais supplémentaires pour la configuration de la simultanéité réservée pour une fonction.
- La simultanéité provisionnée est le nombre d'environnements d'exécution pré-initialisés alloués à votre fonction. Ces environnements d'exécution sont prêts à répondre immédiatement aux demandes de fonctions entrantes. La simultanéité allouée est utile pour réduire les latences de démarrage à froid pour les fonctions et est conçue pour rendre les fonctions disponibles avec des temps de réponse de l'ordre de quelques dixaines de millisecondes. En général, ce sont les charges de travail interactives qui tirent le meilleur parti de cette fonctionnalité. Il s'agit des applications dont les utilisateurs lancent des requêtes, telles que les applications Web et mobiles, et qui sont les plus sensibles à la latence. Les charges de travail asynchrones, telles que les pipelines de traitement des données, sont souvent moins sensibles à la latence et ne nécessitent donc généralement pas de simultanéité allouée. La configuration de la simultanéité provisionnée entraîne des frais supplémentaires sur votre Compte AWS.

Cette rubrique explique comment gérer et configurer la simultanéité provisionnée. Pour une présentation conceptuelle de ces deux types de contrôles de simultanéité, consultez Simultanéité réservée et simultanéité provisionnée. Pour plus d'informations sur la configuration de la simultanéité réservée, consultez the section called "Configuration de la simultanéité réservée".

> **Note**
>
> Les fonctions Lambda liées à un mappage des sources d'événements Amazon MQ ont une simultanéité maximale par défaut. Pour Apache Active MQ, le nombre maximum d'instances simultanées est de 5. Pour Rabbit MQ, le nombre maximum d'instances simultanées est de 1. La définition d'une simultanéité réservée ou provisionnée pour votre fonction ne modifie pas

ces limites. Pour demander une augmentation de la simultanéité maximale par défaut lors de l'utilisation d'Amazon MQ, contactez Support.

Sections

- Configuration de la simultanéité provisionnée
- Estimation précise de la simultanéité provisionnée requise pour une fonction
- Optimisation du code de fonction lors de l'utilisation de la simultanéité provisionnée
- Utilisation de variables d'environnement pour visualiser et contrôler le comportement de simultanéité provisionnée
- Comprendre le comportement de journalisation et de facturation avec la simultanéité provisionnée
- Utilisation d'Application Auto Scaling pour automatiser la gestion de la simultanéité provisionnée

Configuration de la simultanéité provisionnée

Vous pouvez configurer les paramètres de simultanéité provisionnés pour une fonction à l'aide de la console Lambda ou de l'API Lambda.

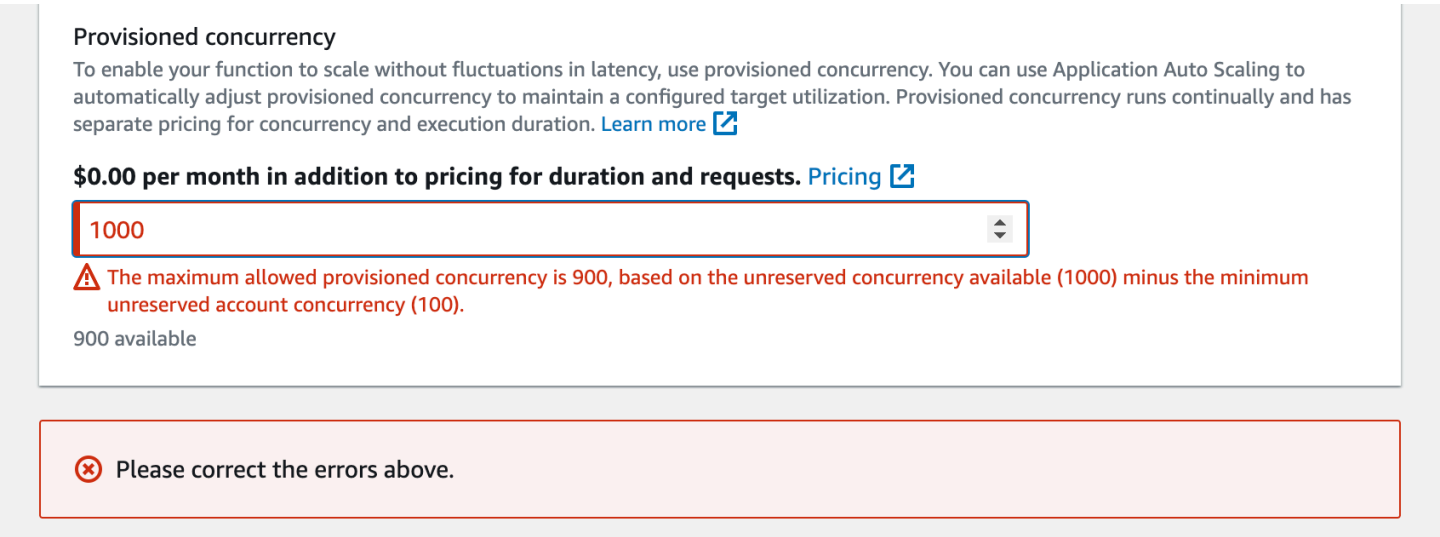
Pour allouer de la simultanéité provisionnée pour une fonction (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction pour laquelle vous souhaitez allouer de la simultanéité provisionnée.
3. Sélectionnez Configuration, puis Concurrency (Simultanéité).
4. Sous Provisioned concurrency configurations (Configurations de simultanéité provisionnée), sélectionnez Add configuration (Ajouter une configuration).
5. Choisissez le type de qualificateur, ainsi que l'alias ou la version.

> **Note**
>
> Vous ne pouvez pas utiliser la simultanéité provisionnée avec la $LATEST version d'une fonction.
> Si votre fonction possède une source d'événement, assurez-vous que la source d'événement pointe vers le bon alias ou la bonne version de la fonction. Sinon, votre fonction n'utilisera pas les environnements de simultanéité provisionnés.

6. Saisissez un nombre sous Simultanéité provisionnée.
7. Choisissez Enregistrer.

Vous pouvez configurer jusqu'à la simultanéité du compte non réservé dans votre compte, moins 100. Les 100 unités de simultanéité restantes concernent les fonctions qui n'utilisent pas la simultanéité réservée. Par exemple, si votre compte a une limite de simultanéité de 1 000 et que vous n'avez pas attribué de simultanéité réservée ou provisionnée à l'une de vos autres fonctions, vous pouvez configurer un maximum de 900 unités de simultanéité provisionnées pour une seule fonction.



La configuration de la simultanéité provisionnée pour une fonction a des conséquences sur le groupe de simultanéité disponible pour d'autres fonctions. Par exemple, si vous configurez 100 unités de simultanéité provisionnée pour `function-a`, les autres fonctions de votre compte doivent partager les 900 unités de simultanéité restantes. Et ce même si `function-a` n'utilise pas les 100 unités.

Il est possible d'allouer à la fois de la simultanéité réservée et de la simultanéité provisionnée pour la même fonction. Dans de tels cas, la simultanéité provisionnée ne peut pas dépasser la simultanéité réservée.

Cette limite s'applique aux versions de fonctions. La simultanéité provisionnée maximale que vous pouvez allouer à une version de fonction spécifique est égale à la simultanéité réservée de la fonction moins la simultanéité provisionnée sur les autres versions de fonction.

Pour configurer la simultanéité provisionnée avec l'API Lambda, utilisez les opérations d'API suivantes.

- PutProvisionedConcurrencyConfig

- GetProvisionedConcurrencyConfig
- ListProvisionedConcurrencyConfigs
- DeleteProvisionedConcurrencyConfig

Par exemple, pour configurer la simultanéité provisionnée avec le AWS Command Line Interface (CLI), utilisez la commande `put-provisioned-concurrency-config`. La commande suivante alloue 100 unités de simultanéité provisionnée pour l'alias `BLUE` d'une fonction nommée `my-function` :

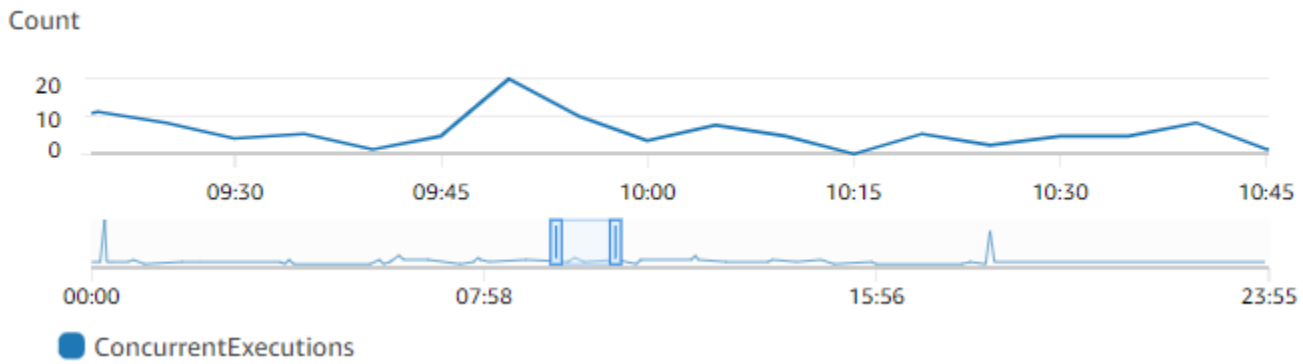
```
aws lambda put-provisioned-concurrency-config --function-name my-function \
  --qualifier BLUE \
  --provisioned-concurrent-executions 100
```

Vous devriez voir une sortie semblable à la suivante :

```
{
  "Requested ProvisionedConcurrentExecutions": 100,
  "Allocated ProvisionedConcurrentExecutions": 0,
  "Status": "IN_PROGRESS",
  "LastModified": "2023-01-21T11:30:00+0000"
}
```

Estimation précise de la simultanéité provisionnée requise pour une fonction

Vous pouvez consulter les métriques de simultanéité de n'importe quelle fonction active à l'aide des métriques CloudWatch. Plus précisément, la métrique `ConcurrentExecutions` vous montre le nombre d'invocations simultanées pour les fonctions de votre compte.



Le graphique précédent indique que cette fonction répond à une moyenne de 5 à 10 demandes simultanées à tout moment, et qu'elle atteint un pic de 20 demandes. Supposons qu'il y ait beaucoup d'autres fonctions dans votre compte. Si cette fonction est essentielle à votre application et que vous avez besoin d'une réponse à faible latence à chaque invocation, configurez au moins 20 unités de simultanéité provisionnée.

Rappelez-vous que vous pouvez également calculer la simultanéité à l'aide de la formule suivante :

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

Pour estimer le niveau de simultanéité dont vous avez besoin, multipliez le nombre moyen de demandes par seconde par la durée moyenne des demandes en secondes. Vous pouvez estimer les demandes moyennes par seconde à l'aide de la métrique `Invocation`, et la durée moyenne des demandes en secondes à l'aide de la métrique `Duration`.

Lors de la configuration de la simultanéité provisionnée, Lambda suggère d'ajouter un tampon de 10 % en plus de la quantité de simultanéité dont votre fonction a généralement besoin. Par exemple, si votre fonction atteint habituellement un pic de 200 demandes simultanées, définissez votre simultanéité provisionnée à 220 (200 demandes simultanées + 10 % = 220 simultanéités provisionnées).

Optimisation du code de fonction lors de l'utilisation de la simultanéité provisionnée

Si vous utilisez la simultanéité provisionnée, pensez à restructurer votre code de fonction pour optimiser la faible latence. Pour les fonctions utilisant la simultanéité provisionnée, Lambda exécute n'importe quel code d'initialisation (c'est-à-dire le chargement de bibliothèques et l'instanciation de clients) au moment de l'allocation. Il est donc conseillé de déplacer un maximum d'initialisation en dehors du gestionnaire de la fonction principale pour éviter d'avoir un impact sur la latence lors des invocations à la fonction. En revanche, l'initialisation de bibliothèques ou l'instanciation de clients dans le code de votre gestionnaire principal signifie que votre fonction doit l'exécuter à chaque fois qu'elle est invoquée (que vous utilisiez ou non la simultanéité provisionnée).

Pour les appels à la demande, Lambda peut avoir besoin de réexécuter votre code d'initialisation chaque fois que votre fonction démarre à froid. Pour de telles fonctions, vous pouvez choisir de différer l'initialisation d'une fonctionnalité spécifique jusqu'à ce que la fonction ait besoin d'elle. Par exemple, prenons le flux de contrôle suivant pour un gestionnaire Lambda :

```
def handler(event, context):
```

```
...
if ( some_condition ):
    // Initialize CLIENT_A to perform a task
else:
    // Do nothing
```

Dans l'exemple précédent, au lieu d'initialiser `CLIENT_A` en dehors du gestionnaire principal, le développeur a initialisé dans l'instruction `if`. Ainsi, Lambda n'exécute ce code que si `some_condition` est satisfaite. Si vous initialisez `CLIENT_A` en dehors du gestionnaire principal, Lambda exécute ce code à chaque démarrage à froid. Cela peut augmenter le temps de latence global.

Vous pouvez mesurer les démarrages à froid à mesure que Lambda prend de l'ampleur en ajoutant la surveillance X-Ray à votre fonction. Une fonction utilisant la simultanéité allouée ne présente pas de comportement de démarrage à froid puisque l'environnement d'exécution est préparé avant l'invocation. Cependant, la simultanéité allouée doit être appliquée à une version ou à un alias spécifique d'une fonction, et non à la version $LATEST. Dans les cas où le comportement de démarrage à froid persiste, assurez-vous que vous invoquez la version de l'alias pour laquelle la simultanéité allouée est configurée.

Utilisation de variables d'environnement pour visualiser et contrôler le comportement de simultanéité provisionnée

Il est possible que votre fonction utilise la totalité de sa simultanéité provisionnée. Lambda utilise des instances à la demande pour gérer tout trafic excédentaire. Pour déterminer quel type d'initialisation Lambda a utilisé pour un environnement particulier, vérifiez la valeur de la variable d'environnement `AWS_LAMBDA_INITIALIZATION_TYPE`. Cette variable a deux valeurs possibles : `provisioned-concurrency` ou`on-demand`. La valeur de `AWS_LAMBDA_INITIALIZATION_TYPE` est immuable et reste constante pendant toute la durée de vie de l'environnement. Pour vérifier la valeur d'une variable d'environnement dans le code de votre fonction, consultez Récupération de variables d'environnement Lambda.

Si vous utilisez l'environnement d'exécution .NET 8, vous pouvez configurer la variable d'environnement `AWS_LAMBDA_DOTNET_PREJIT` pour améliorer la latence des fonctions, même si elles n'utilisent pas de simultanéité provisionnée. Le runtime .NET compile et initialise lentement chaque bibliothèque que votre code appelle pour la première fois. Par conséquent, la première invocation d'une fonction Lambda peut prendre plus de temps que les invocations suivantes. Pour atténuer ce problème, vous pouvez choisir l'une des trois valeurs `AWS_LAMBDA_DOTNET_PREJIT` :

- `ProvisionedConcurrency` : Lambda effectue une compilation JIT à l'avance pour tous les environnements utilisant la simultanéité provisionnée. C'est la valeur par défaut.
- `Always` : Lambda effectue une compilation JIT à l'avance pour chaque environnement, même si la fonction n'utilise pas la simultanéité provisionnée.
- `Never` : Lambda désactive la compilation JIT à l'avance pour tous les environnements.

Comprendre le comportement de journalisation et de facturation avec la simultanéité provisionnée

Pour les environnements de simultanéité provisionnée, le code d'initialisation de votre fonction s'exécute pendant l'allocation et périodiquement lorsque Lambda recycle les instances actives de votre environnement. Lambda vous facture l'initialisation même si l'instance d'environnement ne traite jamais de demande. La simultanéité provisionnée s'exécute en continu et est facturée séparément des coûts d'initialisation et d'invocation. Pour plus d'informations, consultez Tarification AWS Lambda.

Lorsque vous configurez une fonction Lambda avec une simultanéité provisionnée, Lambda pré-initialise cet environnement d'exécution afin qu'il soit disponible avant les demandes d'invocation. Lambda enregistre le champ Init Duration de la fonction dans un événement de journal platform-initReport au format de journalisation JSON chaque fois que l'environnement est initialisé. Pour voir cet événement de journal, configurez votre Niveau de journal JSON sur au moins `INFO`. Vous pouvez également utiliser l'API de télémétrie pour consulter les événements de plateforme dans lesquels le champ Init Duration est indiqué.

Utilisation d'Application Auto Scaling pour automatiser la gestion de la simultanéité provisionnée

Vous pouvez utiliser Application Auto Scaling pour gérer la simultanéité provisionnée selon une planification ou en fonction de l'utilisation. Si vous observez des schémas prévisibles de trafic vers votre fonction, utilisez la mise à l'échelle programmée. Si vous souhaitez que votre fonction maintienne un pourcentage d'utilisation spécifique, utilisez une politique de mise à l'échelle de suivi cible.

> **Note**
>
> Si vous utilisez Application Auto Scaling pour gérer la simultanéité provisionnée de votre fonction, assurez-vous de configurer d'abord une valeur de simultanéité provisionnée initiale.

Si votre fonction ne possède pas de valeur de simultanéité provisionnée initiale, Application Auto Scaling risque de ne pas gérer correctement la mise à l'échelle des fonctions.

Mise à l'échelle planifiée

Avec Application Auto Scaling, vous pouvez définir votre propre planification de mise à l'échelle en fonction des changements de charge prévisibles. Pour plus d'informations et d'exemples, consultez Mise à l'échelle programmée pour Application Auto Scaling dans le Guide de l'utilisateur Application Auto Scaling, et Scheduling AWS Lambda Provisioned Concurrency for recurring peak usage sur le blog AWS Compute.

Suivi de la cible

Avec le suivi des cibles, Application Auto Scaling crée et gère un ensemble d'alarmes CloudWatch basées sur la façon dont vous définissez votre politique de mise à l'échelle. Lorsque ces alarmes sont activées, Application Auto Scaling ajuste automatiquement la quantité d'environnements alloués à l'aide de la simultanéité provisionnée. Le suivi des cibles est idéal pour les applications dont les modèles de trafic ne sont pas prévisibles.

Pour mettre à l'échelle la simultanéité provisionnée à l'aide du suivi des cibles, utilisez les opérations de l'API Application Auto Scaling `RegisterScalableTarget` et `PutScalingPolicy`. Par exemple, si vous utilisez la AWS Command Line Interface (CLI), procédez comme suit :

1. Enregistrez l'alias d'une fonction en tant que cible de mise à l'échelle. L'exemple suivant enregistre l'alias BLUE d'une fonction nommée `my-function` :

```
aws application-autoscaling register-scalable-target --service-namespace lambda \
    --resource-id function:my-function:BLUE --min-capacity 1 --max-capacity 100 \
    --scalable-dimension lambda:function:ProvisionedConcurrency
```

2. Ensuite, appliquez une stratégie de mise à l'échelle à la cible. L'exemple suivant configure Application Auto Scaling afin d'ajuster la configuration de simultanéité provisionnée pour un alias de façon à maintenir l'utilisation proche de 70 %, mais vous pouvez appliquer n'importe quelle valeur comprise entre 10 % et 90 %.

```
aws application-autoscaling put-scaling-policy \
    --service-namespace lambda \
    --scalable-dimension lambda:function:ProvisionedConcurrency \
    --resource-id function:my-function:BLUE \
```

```
    --policy-name my-policy \
    --policy-type TargetTrackingScaling \
    --target-tracking-scaling-policy-configuration '{ "TargetValue":
 0.7, "PredefinedMetricSpecification": { "PredefinedMetricType":
 "LambdaProvisionedConcurrencyUtilization" }}'
```

Vous devriez obtenir un résultat du type suivant :

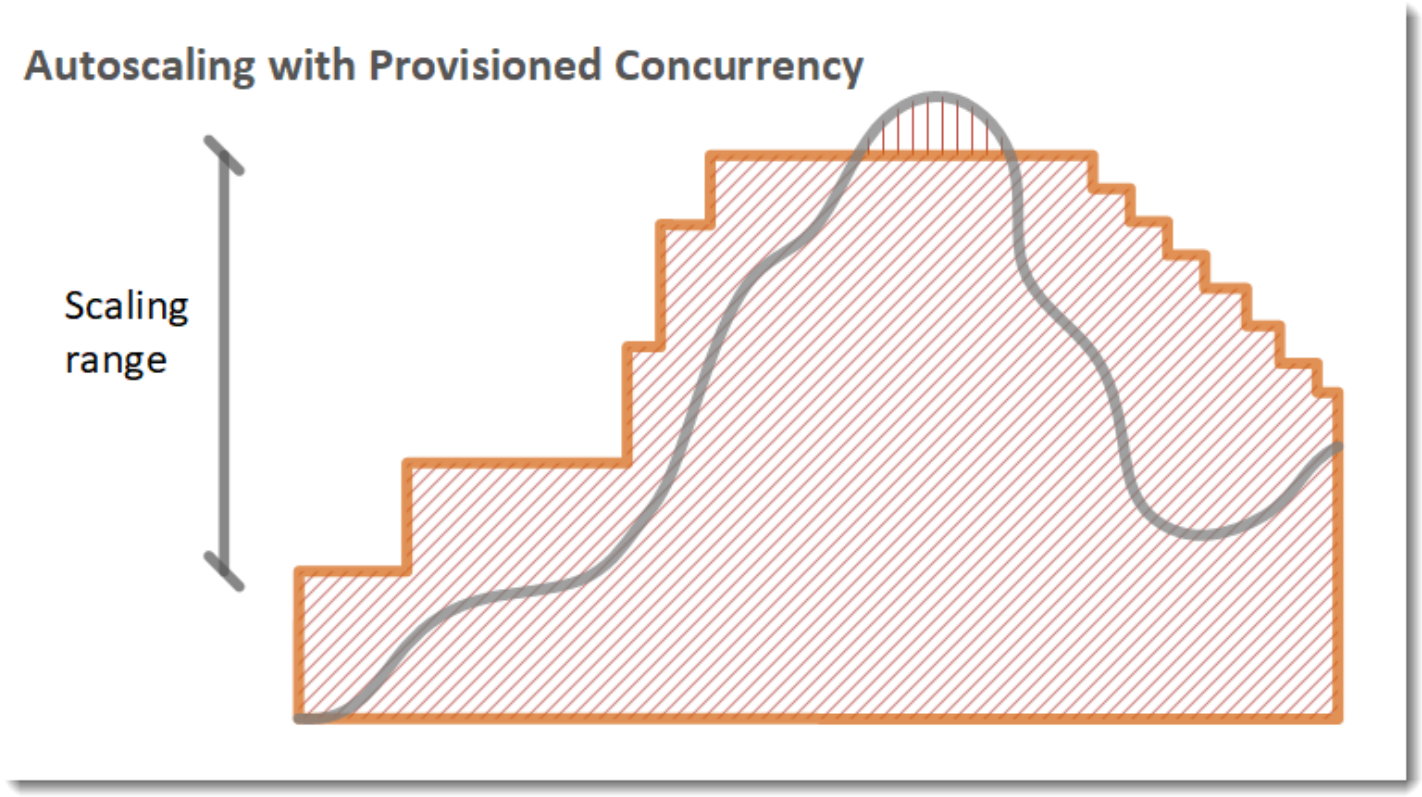
```
{
    "PolicyARN": "arn:aws:autoscaling:us-
east-2:123456789012:scalingPolicy:12266dbb-1524-xmpl-a64e-9a0a34b996fa:resource/lambda/
function:my-function:BLUE:policyName/my-policy",
    "Alarms": [
        {
            "AlarmName": "TargetTracking-function:my-function:BLUE-AlarmHigh-aed0e274-
xmpl-40fe-8cba-2e78f000c0a7",
            "AlarmARN": "arn:aws:cloudwatch:us-
east-2:123456789012:alarm:TargetTracking-function:my-function:BLUE-AlarmHigh-aed0e274-
xmpl-40fe-8cba-2e78f000c0a7"
        },
        {
            "AlarmName": "TargetTracking-function:my-function:BLUE-AlarmLow-7e1a928e-
xmpl-4d2b-8c01-782321bc6f66",
            "AlarmARN": "arn:aws:cloudwatch:us-
east-2:123456789012:alarm:TargetTracking-function:my-function:BLUE-AlarmLow-7e1a928e-
xmpl-4d2b-8c01-782321bc6f66"
        }
    ]
}
```

Application Auto Scaling crée deux alarmes dans CloudWatch. La première alarme se déclenche lorsque l'utilisation de la simultanéité provisionnée dépasse systématiquement 70 %. Lorsque cela se produit, Application Auto Scaling alloue davantage de simultanéité approvisionnée afin de réduire l'utilisation. La deuxième alarme se déclenche lorsque l'utilisation est constamment inférieure à 63 % (90 % de la cible de 70 %). Lorsque cela se produit, Application Auto Scaling réduit la simultanéité approvisionnée de l'alias.

> **Note**
>
> Lambda émet la métrique `ProvisionedConcurrencyUtilization` uniquement lorsque votre fonction est active et reçoit des demandes. Pendant les périodes d'inactivité, aucune

métrique n'est émise et vos alarmes d'autoscaling entrent dans l'état `INSUFFICIENT_DATA`. Par conséquent, l'autoscaling d'application ne pourra pas ajuster la simultanéité allouée à votre fonction. Cela peut entraîner des frais imprévus.

Dans l'exemple suivant, une fonction adapte son échelle entre une quantité minimum et maximum de simultanéité approvisionnée en fonction de l'utilisation.



Légende

- Instances de la fonction
- Demandes ouvertes
- Simultanéité provisionnée
- Simultanéité standard

Quand le nombre de demandes ouvertes augmente, Application Auto Scaling augmente la simultanéité provisionnée par échelons jusqu'à ce qu'elle atteigne le maximum configuré. Une fois le maximum atteint, la fonction peut continuer à se mettre à l'échelle en fonction de la simultanéité standard, non réservée, si votre compte n'a pas atteint sa limite de simultanéité. Lorsque l'utilisation chute et reste constamment faible, Application Auto Scaling réduit la simultanéité provisionnée par échelons périodiques plus petits.

Les deux alarmes gérées par Application Auto Scaling utilisent la statistique moyenne par défaut. Les fonctions qui subissent des rafales de trafic peuvent ne pas déclencher ces alarmes. Par exemple, supposons que votre fonction Lambda s'exécute rapidement (c'est-à-dire entre 20 et 100 ms) et que votre modèle de trafic se produise sous forme de rafales rapides. Dans ce cas, le nombre de demandes dépasse la simultanéité allouée pendant la rafale. Cependant, Application Auto Scaling nécessite que la charge en rafale soit maintenue pendant au moins 3 minutes afin de provisionner des environnements supplémentaires. De plus, les deux alarmes CloudWatch nécessitent trois points de données qui atteignent la moyenne cible avant d'activer la politique d'autoscaling. Si votre fonction connaît des pics de trafic rapides, l'utilisation de la statistique Maximum au lieu de la statistique Average peut être plus efficace pour mettre à l'échelle la simultanéité provisionnée afin de minimiser les démarrages à froid.

Pour plus d'informations sur l'utilisation des politiques de mise à l'échelle du suivi des cibles, consultez Politiques de mise à l'échelle du suivi des cibles pour Application Auto Scaling.

Comportement de mise à l'échelle Lambda

Au fur et à mesure que votre fonction reçoit des demandes, Lambda augmente automatiquement le nombre d'environnements d'exécution pour traiter ces demandes jusqu'à ce que votre compte atteigne sa limite de simultanéité. Toutefois, pour éviter une mise à l'échelle excessive en réponse à des pics de trafic soudains, Lambda limite la rapidité avec laquelle vos fonctions peuvent être mises à l'échelle. Ce taux de mise à l'échelle de la simultanéité est la vitesse maximale à laquelle les fonctions de votre compte peuvent se mettre à l'échelle en réponse à l'augmentation du nombre de requêtes. (C'est-à-dire la rapidité avec laquelle Lambda peut créer de nouveaux environnements d'exécution.) Le aux de mise à l'échelle de la simultanéité est différent de la limite de simultanéité au niveau du compte, qui est le montant total de simultanéité disponible pour vos fonctions.

Taux de mise à l'échelle de la simultanéité

Dans chaque Région AWS et pour chaque fonction, votre taux de mise à l'échelle de la simultanéité est de 1 000 instances d'environnement d'exécution toutes les 10 secondes (ou 10 000 requêtes par seconde toutes les 10 secondes). En d'autres termes, toutes les 10 secondes, Lambda peut allouer au maximum 1 000 instances d'environnement d'exécution supplémentaires, ou accueillir 10 000 requêtes supplémentaires par seconde, pour chacune de vos fonctions.

En général, il n'est pas nécessaire de se soucier de cette limitation. La vitesse de mise à l'échelle de Lambda est suffisante dans la plupart des cas d'utilisation.

Il est important de noter que le taux de mise à l'échelle de la simultanéité est une limite au niveau de la fonction. Cela signifie que chaque fonction de votre compte peut être mise à l'échelle indépendamment des autres fonctions.

> **Note**
>
> Dans la pratique, Lambda s'efforce d'augmenter votre taux de simultanéité de manière continue au fil du temps, plutôt que de procéder à une seule recharge de 1 000 unités toutes les 10 secondes.

Lambda n'accumule pas les portions inutilisées de votre taux de mise à l'échelle de la simultanéité. Cela signifie qu'à tout moment, votre taux de mise à l'échelle est toujours de 1 000 unités simultanées au maximum. Par exemple, si vous n'utilisez aucune de vos 1 000 unités de simultanéité disponibles dans un intervalle de 10 secondes, vous n'accumulerez pas 1 000 unités

supplémentaires dans le prochain intervalle de 10 secondes. Votre taux de mise à l'échelle de la simultanéité est toujours de 1 000 dans le prochain intervalle de 10 secondes.

Tant que votre fonction continue de recevoir un nombre croissant de demandes, Lambda est mis à l'échelle au rythme le plus rapide à votre disposition, jusqu'à la limite de simultanéité de votre compte. Vous pouvez limiter le niveau de simultanéité que les fonctions individuelles peuvent utiliser en configurant la simultanéité réservée. Si l'entrée des demandes est plus rapide que la capacité de mise à l'échelle de votre fonction ou si votre fonction atteint la simultanéité maximale, des demandes supplémentaires échouent alors avec un code de limitation (code d'état 429).

Surveillance de la simultanéité

Lambda émet des métriques Amazon CloudWatch pour vous aider à surveiller la simultanéité de vos fonctions. Cette rubrique explique ces métriques et comment les interpréter.

Sections

- Métriques de simultanéité générales
- Métriques de simultanéité provisionnée
- Travailler avec la métrique ClaimedAccountConcurrency

Métriques de simultanéité générales

Utilisez les métriques suivantes pour surveiller la simultanéité de vos fonctions Lambda. La granularité de chaque métrique est d'une minute.

- `ConcurrentExecutions` : le nombre d'invocations simultanées actives à un moment donné. Lambda émet cette métrique pour toutes les fonctions, versions et alias. Pour toute fonction de la console Lambda, Lambda affiche le graphique pour `ConcurrentExecutions` nativement native dans l'onglet Surveillance, sous Métriques. Consultez cette métrique en utilisant MAX.
- `UnreservedConcurrentExecutions` : le nombre d'invocations simultanées actives qui utilisent la simultanéité non réservée. Lambda émet cette métrique pour toutes les fonctions d'une région. Consultez cette métrique en utilisant MAX.
- `ClaimedAccountConcurrency` – Le niveau de simultanéité qui n'est pas disponible pour les appels à la demande. `ClaimedAccountConcurrency` est égal à `UnreservedConcurrentExecutions` plus le montant de la simultanéité allouée (c'est-à-dire le total de la simultanéité réservée plus le total de la simultanéité provisionnée). Si `ClaimedAccountConcurrency` dépasse la limite de simultanéité de votre compte, vous pouvez demander une limite de simultanéité de compte plus élevée. Consultez cette métrique en utilisant MAX. Pour de plus amples informations, consultez Travailler avec la métrique `ClaimedAccountConcurrency`.

Métriques de simultanéité provisionnée

Utilisez les métriques suivantes pour surveiller les fonctions Lambda utilisant la simultanéité provisionnée. La granularité de chaque métrique est d'une minute.

- `ProvisionedConcurrentExecutions` : le nombre d'instances d'environnement d'exécution qui traitent activement une invocation sur une simultanéité allouée. Lambda émet cette métrique pour chaque version de fonction et alias avec simultanéité provisionnée configurée. Consultez cette métrique en utilisant MAX.

`ProvisionedConcurrentExecutions` n'est pas le même que le nombre total de simultanéités provisionnées que vous allouez. Par exemple, supposons que vous allouiez 100 unités de simultanéité provisionnée à une version de fonction. Au cours d'une minute donnée, si au maximum 50 de ces 100 environnements d'exécution traitent des invocations simultanément, la valeur de MAX (`ProvisionedConcurrentExecutions`) est de 50.

- `ProvisionedConcurrencyInvocations` – Le nombre de fois où Lambda invoque votre code de fonction en utilisant la simultanéité provisionnée. Lambda émet cette métrique pour chaque version de fonction et alias avec simultanéité provisionnée configurée. Consultez cette métrique en utilisant SUM.

`ProvisionedConcurrencyInvocations` diffère de `ProvisionedConcurrentExecutions`, car `ProvisionedConcurrencyInvocations` compte le nombre total d'invocations, tandis que `ProvisionedConcurrentExecutions` compte le nombre d'environnements actifs. Pour comprendre cette distinction, examinons le scénario suivant :

Dans cet exemple, supposons que vous receviez une invocation par minute et que chaque invocation dure deux minutes. Chaque barre horizontale orange représente une demande unique. Supposons que vous allouiez dix unités de simultanéité provisionnée à cette fonction, de sorte que chaque demande s'exécute sur la simultanéité provisionnée.

Entre les minutes 0 et 1, `Request 1` entre en jeu. À la minute 1, la valeur de MAX (`ProvisionedConcurrentExecutions`) est de un, puisqu'au maximum un environnement d'exécution a été actif au cours de la dernière minute. La valeur de SUM (`ProvisionedConcurrencyInvocations`) est également de un, puisqu'une nouvelle demande a été reçue au cours de la dernière minute.

Entre les minutes 1 et 2, `Request 2` entre en jeu et `Request 1` continue de fonctionner. À la minute 2, la valeur de MAX (`ProvisionedConcurrentExecutions`) est de deux, puisqu'au maximum deux environnements d'exécution ont été actifs au cours de la dernière minute. Toutefois, la valeur de SUM (`ProvisionedConcurrencyInvocations`) est de un, car une seule nouvelle demande a été reçue au cours de la dernière minute. Ce comportement métrique se poursuit jusqu'à la fin de l'exemple.

- `ProvisionedConcurrencySpilloverInvocations` : le nombre de fois où Lambda invoque votre fonction sur la simultanéité standard (réservée ou non) lorsque toute la simultanéité provisionnée est utilisée. Lambda émet cette métrique pour chaque version de fonction et alias avec simultanéité provisionnée configurée. Consultez cette métrique en utilisant SUM. La valeur de `ProvisionedConcurrencyInvocations` + `ProvisionedConcurrencySpilloverInvocations` doit être égale au nombre total d'invocations de fonctions (c'est-à-dire à la métrique `Invocations`).

  `ProvisionedConcurrencyUtilization` : le pourcentage de la simultanéité provisionnée utilisée (c'est à dire la valeur de `ProvisionedConcurrentExecutions` divisée par la quantité totale de la simultanéité provisionnée allouée). Lambda émet cette métrique pour chaque version de fonction et alias avec simultanéité provisionnée configurée. Consultez cette métrique en utilisant MAX.

Par exemple, supposons que vous provisionniez 100 unités de simultanéité provisionnée à une version de fonction. Au cours d'une minute donnée, si au maximum 60 de ces 100 environnements d'exécution traitent des invocations simultanément, la valeur de MAX (`ProvisionedConcurrentExecutions`) est de 60 et la valeur de MAX (`ProvisionedConcurrencyUtilization`) est de 0,6.

Une valeur élevée pour `ProvisionedConcurrencySpilloverInvocations` peut indiquer que vous devez allouer un accès simultané supplémentaire à votre fonction. Vous pouvez également configurer Application Auto Scaling pour gérer la mise à l'échelle automatique de la simultanéité provisionnée en fonction de seuils prédéfinis.

À l'inverse, des valeurs constamment faibles pour `ProvisionedConcurrencyUtilization` peuvent indiquer que vous avez trop alloué de simultanéité provisionnée pour votre fonction.

Travailler avec la métrique **`ClaimedAccountConcurrency`**

Lambda utilise la métrique `ClaimedAccountConcurrency` pour déterminer le niveau de simultanéité disponible sur votre compte pour les appels à la demande. Lambda calcule `ClaimedAccountConcurrency` à l'aide de la formule suivante :

```
ClaimedAccountConcurrency = UnreservedConcurrentExecutions + (allocated concurrency)
```

`UnreservedConcurrentExecutions` correspond au nombre d'invocations simultanées actives qui utilisent la simultanéité non réservée. La simultanéité allouée est la somme des deux parties suivantes (en les remplaçant RC en tant que « simultanéité réservée » et PC en tant que « simultanéité provisionnée ») :

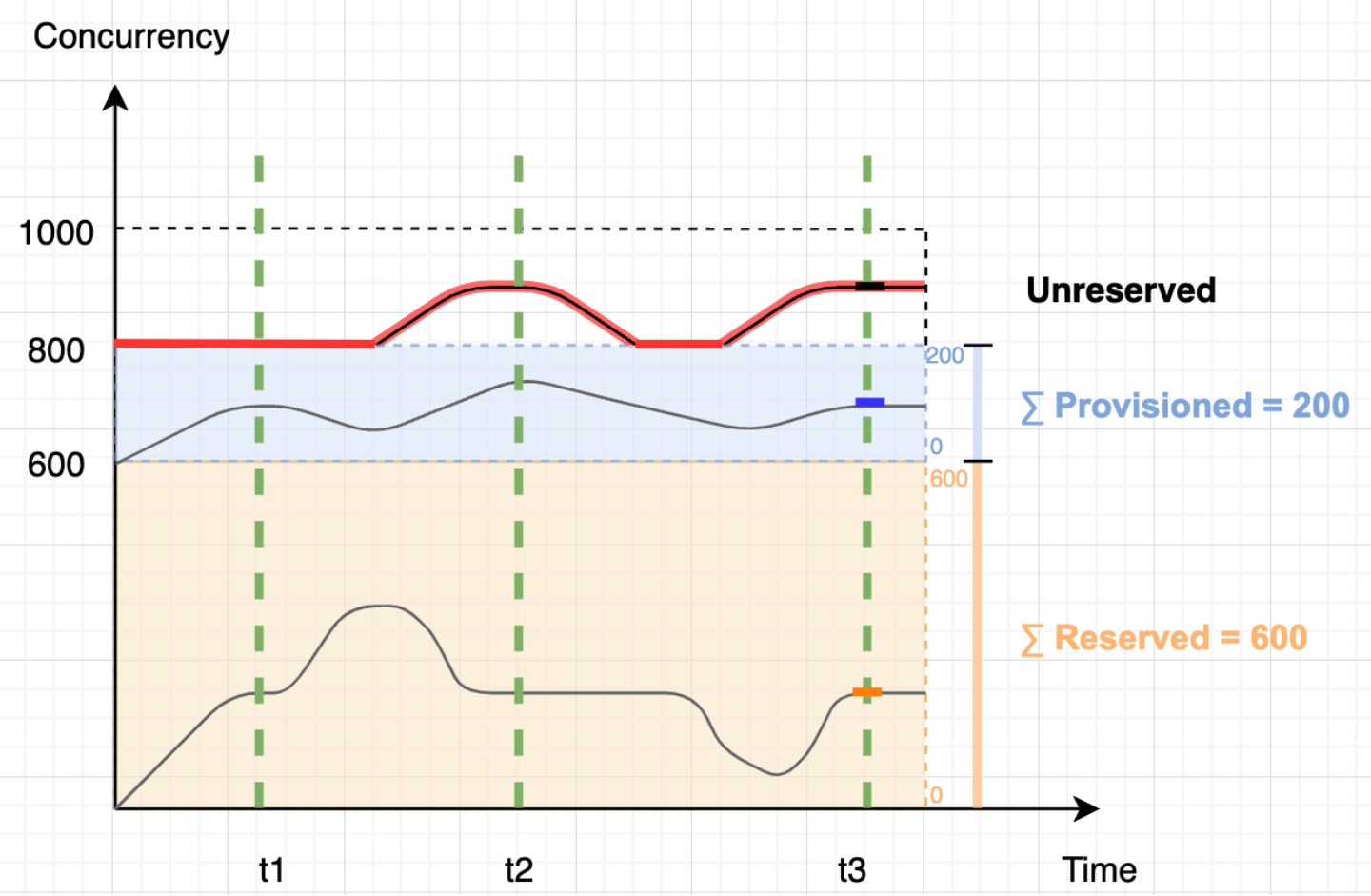
- Total pour RC entre toutes les fonctions d'une région.
- Total de PC entre toutes les fonctions d'une région qui utilisent PC, à l'exception des fonctions qui utilisent RC.

> **Note**
>
> Vous ne pouvez pas allouer plus de PC que ce RC pour une fonction. Ainsi, le RC d'une fonction est toujours supérieur ou égal à son PC. Pour calculer la contribution à la simultanéité allouée pour de telles fonctions avec PC et RC, Lambda ne prend en compte que RC, qui est le maximum des deux.

Lambda utilise la métrique `ClaimedAccountConcurrency` plutôt que `ConcurrentExecutions` pour déterminer le niveau de simultanéité disponible pour les appels à la demande. Bien que la métrique `ConcurrentExecutions` soit utile pour suivre le nombre d'appels simultanés actifs, elle ne reflète pas toujours votre véritable disponibilité simultanée. En effet, Lambda prend également en compte la simultanéité réservée et la simultanéité provisionnée pour déterminer la disponibilité.

Pour illustrer `ClaimedAccountConcurrency`, imaginez un scénario dans lequel vous configurez une grande partie de la simultanéité réservée et de la simultanéité provisionnée entre vos fonctions qui restent largement inutilisées. Dans l'exemple suivant, supposons que la limite de simultanéité de

votre compte est de 1 000 et que votre compte comporte deux fonctions principales : `function-orange` et `function-blue`. Vous allouez 600 unités de simultanéité réservée pour `function-orange`. Vous allouez 200 unités de simultanéité provisionnée pour `function-blue`. Supposons qu'au fil du temps, vous déployiez des fonctions supplémentaires et observiez le schéma de trafic suivant :



Dans le schéma précédent, les lignes noires indiquent l'utilisation simultanée réelle au fil du temps, et la ligne rouge indique la valeur de `ClaimedAccountConcurrency` au fil du temps. Dans ce scénario, `ClaimedAccountConcurrency` est égal à 800 au minimum malgré une faible utilisation réelle de la simultanéité entre vos fonctions. Cela est dû au fait que vous avez alloué 800 unités de simultanéité au total pour `function-orange` et `function-blue`. Du point de vue de Lambda, vous avez « revendiqué » cette simultanéité pour l'utiliser, il ne vous reste donc que 200 unités de simultanéité pour les autres fonctions.

Pour ce scénario, la simultanéité allouée est de 800 dans la formule `ClaimedAccountConcurrency`. Nous pouvons ensuite déduire la valeur de `ClaimedAccountConcurrency` en différents points du diagramme :

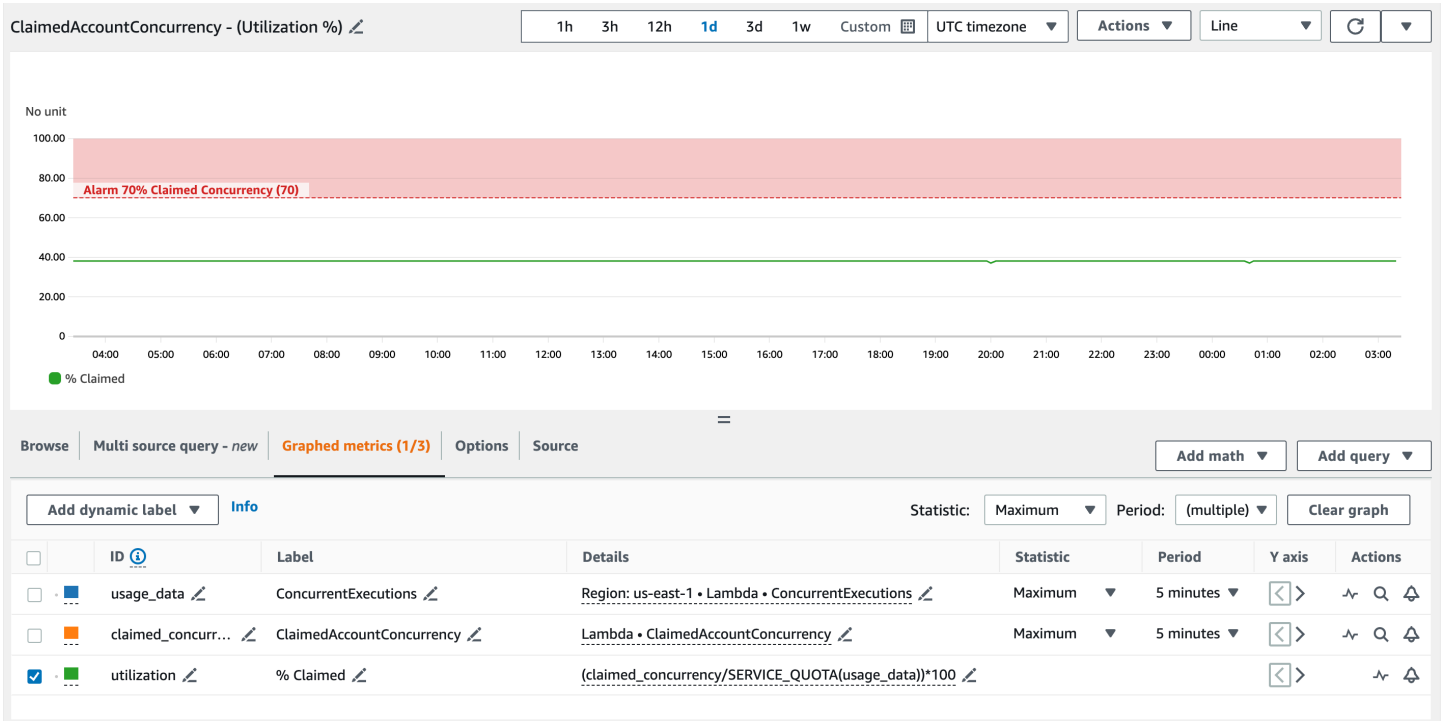
- Au niveau de `t1`, `ClaimedAccountConcurrency` est égal à 800 (800 + 0 `UnreservedConcurrentExecutions`).
- Au niveau de `t2`, `ClaimedAccountConcurrency` est égal à 900 (800 + 100 `UnreservedConcurrentExecutions`).
- Au niveau de `t3`, `ClaimedAccountConcurrency` est à nouveau égal à 900 (800 + 100 `UnreservedConcurrentExecutions`).

Configuration de la métrique **`ClaimedAccountConcurrency`** dans CloudWatch

Lambda émet la métrique `ClaimedAccountConcurrency` dans CloudWatch. Utilisez cette métrique avec la valeur de `SERVICE_QUOTA(ConcurrentExecutions)` pour obtenir le pourcentage d'utilisation de la simultanéité sur votre compte, comme indiqué dans la formule suivante :

```
Utilization = (ClaimedAccountConcurrency/SERVICE_QUOTA(ConcurrentExecutions)) * 100%
```

La capture d'écran suivante montre comment représenter graphiquement cette formule dans CloudWatch. La ligne verte `claim_utilization` représente l'utilisation simultanée de ce compte, qui est d'environ 40 % :



La capture d'écran précédente inclut également une alarme CloudWatch qui active l'état `ALARM` lorsque l'utilisation simultanée dépasse 70 %. Vous pouvez utiliser la métrique `ClaimedAccountConcurrency` ainsi que des alarmes similaires pour déterminer de manière proactive le moment où vous pourriez avoir besoin de demander une limite de simultanéité de comptes plus élevée.

Création de fonctions Lambda avec Node.js

Vous pouvez exécuter JavaScript du code avec Node.js dans AWS Lambda. Lambda fournit des environnements d'exécution pour Node.js, qui exécutent votre code afin de traiter des événements. Votre code s'exécute dans un environnement qui inclut le AWS SDK pour JavaScript, avec les informations d'identification d'un rôle Gestion des identités et des accès AWS (IAM) que vous gérez. Pour en savoir plus sur les versions du kit SDK incluses dans les environnements d'exécution Node.js, consultez the section called "Versions du SDK incluses dans l'environnement d'exécution".

Lambda prend en charge les environnements d'exécution Node.js suivants.

Name	Identifiant	Système d'exploitation	Date d'obsolescence	Créer la fonction de blocage	Mettre à jour la fonction de blocage
Node.js 24	`nodejs24.x`	Amazon Linux 2	30 avril 2028	1 juin 2028	1 juillet 2028
Node.js 22	`nodejs22.x`	Amazon Linux 2	30 avril 2027	1 juin 2027	1 juillet 2027
Node.js 20	`nodejs20.x`	Amazon Linux 2	30 avril 2026	31 août 2026	30 sept. 2026

Pour créer une fonction Node.js.

1. Ouvrez la console Lambda.
2. Sélectionnez Créer une fonction.
3. Configurez les paramètres suivants :
   - Nom de la fonction : saisissez le nom de la fonction.
   - Exécution : Choisissez Node.js 24.x.
4. Choisissez Créer une fonction.

La console crée une fonction Lambda avec un seul fichier source nommé `index.mjs`. Vous pouvez modifier ce fichier et ajouter d'autres fichiers dans l'éditeur de code intégré. Dans la section

DÉPLOYER, choisissez Déployer pour mettre à jour le code de votre fonction. Ensuite, pour exécuter votre code, choisissez Créer un événement de test dans la section ÉVÉNEMENTS DE TEST.

Le fichier `index.mjs` exporte une fonction nommée `handler` qui accepte un objet événement et un objet contexte. Il s'agit de la fonction de gestionnaire que Lambda appelle lors de l'appel de la fonction. Le runtime de la fonction Node.js obtient des événements d'invocations à partir de Lambda et les transmet au gestionnaire. Dans la configuration de fonction, la valeur de gestionnaire est `index.handler`.

Lorsque vous enregistrez votre code de fonction, la console Lambda crée un package de déploiement d'archive de fichiers .zip. Lorsque vous développez votre code de fonction en dehors de la console (à l'aide d'un IDE), vous devez créer un package de déploiement pour charger votre code dans la fonction Lambda.

Le runtime de la fonction transmet un objet de contexte au gestionnaire, en plus de l'événement d'invocation. L'objet de contexte contient des informations supplémentaires sur l'invocation, la fonction et l'environnement d'exécution. Des informations supplémentaires sont disponibles dans les variables d'environnement.

Votre fonction Lambda est fournie avec un groupe de CloudWatch journaux Logs. La fonction runtime envoie les détails de chaque appel à CloudWatch Logs. Il relaie tous les journaux que votre fonction génère pendant l'invocation. Si votre fonction renvoie une erreur, Lambda met en forme l'erreur et la renvoie à l'appelant.

Rubriques

- Versions du SDK incluses dans l'environnement d'exécution
- Utilisation de keep-alive pour les connexions TCP
- Chargement des certificats CA
- Fonctionnalités expérimentales de Node.js
- Définition du gestionnaire de fonction Lambda dans Node.js
- Déployer des fonctions Lambda en Node.js avec des archives de fichiers .zip
- Déployer des fonctions Lambda en Node.js avec des images conteneurs
- Utilisation de couches pour les fonctions Lambda Node.js
- Utilisation de l'objet de contexte Lambda pour récupérer les informations de la fonction Node.js
- Journalisation et surveillance des fonctions Lambda Node.js
- Instrumentation du code Node.js dans AWS Lambda

Versions du SDK incluses dans l'environnement d'exécution

Tous les environnements d'exécution Lambda Node.js pris en charge incluent une version mineure spécifique de la AWS SDK pour JavaScript v3, et non la dernière version. La version mineure spécifique incluse dans l'environnement d'exécution dépend de la version de l'environnement d'exécution et de votre Région AWS. Pour trouver la version spécifique du kit SDK incluse dans l'environnement d'exécution que vous utilisez, créez une fonction Lambda avec le code suivant.

Example index.mjs

```
import packageJson from '@aws-sdk/client-s3/package.json' with { type: 'json' };

export const handler = async () => ({ version: packageJson.version });
```

Ceci renvoie une réponse au format suivant :

```
{
  "version": "3.632.0"
}
```

Pour de plus amples informations, veuillez consulter Utilisation du SDK pour la JavaScript version 3 dans votre gestionnaire.

Utilisation de keep-alive pour les connexions TCP

L' HTTP/HTTPS agent Node.js par défaut crée une nouvelle connexion TCP pour chaque nouvelle demande. Pour éviter le coût lié à l'établissement de nouvelles connexions, keep-alive est activé par défaut dans tous les environnements d'exécution Node.js pris en charge. Keep-alive peut réduire les temps de requête pour les fonctions Lambda qui effectuent plusieurs appels d'API à l'aide du kit SDK.

Pour désactiver keep-alive, consultez la section Réutilisation des connexions avec keep-alive dans le fichier Node.js du guide du développeur du AWS SDK pour 3.x. JavaScript Pour plus d'informations sur l'utilisation de keep-alive, voir HTTP keep-alive est activé par défaut dans le AWS SDK modulaire ou sur le blog des outils de JavaScript développement. AWS

Chargement des certificats CA

Pour les versions de l'environnement d'exécution Node.js antérieures à Node.js 18, Lambda charge automatiquement les certificats CA (autorité de certification) spécifiques à Amazon afin de vous

permettre de créer plus facilement des fonctions qui interagissent avec d'autres Services AWS. Par exemple, Lambda inclut les certificats Amazon RDS nécessaires pour valider le certificat d'identité du serveur installé sur votre base de données Amazon RDS. Ce comportement peut avoir un impact sur les performances lors des démarrages à froid.

À partir de Node.js 20, Lambda ne charge plus de certificats CA supplémentaires par défaut. L'exécution Node.js 20 contient un fichier de certificat contenant tous les certificats Amazon CA situés à l'adresse `/var/runtime/ca-cert.pem`. Pour restaurer le même comportement à partir de Node.js 18 et exécutions antérieures, définissez la variable d'environnement `NODE_EXTRA_CA_CERTS`sur `/var/runtime/ca-cert.pem`.

Pour des performances optimales, nous vous recommandons d'effectuer la création d'une offre groupée uniquement pour les certificats dont vous avez besoin avec votre package de déploiement et de les charger via la variable d'environnement `NODE_EXTRA_CA_CERTS`. Le fichier de certificats doit être composé d'un ou de plusieurs certificats d'autorité de certification racine ou intermédiaire sécurisés au format PEM. Par exemple, pour RDS, incluez les certificats requis à côté de votre code en tant que `certificates/rds.pem`. Chargez ensuite les certificats en réglant `NODE_EXTRA_CA_CERTS` sur `/var/task/certificates/rds.pem`.

Fonctionnalités expérimentales de Node.js

Les versions en amont du langage Node.js activent certaines fonctionnalités expérimentales par défaut. Lambda désactive ces fonctionnalités pour garantir la stabilité de l'exécution et des performances constantes. Le tableau suivant répertorie les fonctionnalités expérimentales désactivées par Lambda.

Fonctionnalité expérimentale	Versions de Node.js prises en charge	Indicateur Node.js appliqué par Lambda	Indicateur Lambda à réactiver
Prise en charge de l'importation de modules à l'aide de require dans les modules ES	Node.js 20, Node.js 22	`--no-experimental-require-module`	`--experimental-require-module`
Prise en charge de la détection automatique des modules ES par	Node.js 22	`--no-experimental-detect-module`	`--experimental-detect-module`

Fonctionnalité expérimentale	Versions de Node.js prises en charge	Indicateur Node.js appliqué par Lambda	Indicateur Lambda à réactiver
rapport aux modules CommonJS			

Pour activer une fonctionnalité expérimentale désactivée, définissez l'indicateur de réactivation dans la variable d'environnement `NODE_OPTIONS`. Par exemple, pour activer la prise en charge de require dans les modules ES, réglez `NODE_OPTIONS` sur `--experimental-require-module`. Lambda détecte ce remplacement et supprime l'indicateur de désactivation correspondant.

> **Important**
>
> L'utilisation de fonctionnalités expérimentales peut entraîner des problèmes d'instabilité et de performance. Ces fonctionnalités sont susceptibles d'être modifiées ou supprimées dans les futures versions de Node.js. Les fonctions qui utilisent des fonctionnalités expérimentales ne sont pas éligibles au contrat de niveau de service (SLA) Lambda ou à AWS Support

Définition du gestionnaire de fonction Lambda dans Node.js

Le gestionnaire de fonction Lambda est la méthode dans votre code de fonction qui traite les événements. Lorsque votre fonction est invoquée, Lambda exécute la méthode du gestionnaire. Votre fonction s'exécute jusqu'à ce que le gestionnaire renvoie une réponse, se ferme ou expire.

Cette page explique comment utiliser les gestionnaires de fonctions Lambda dans Node.js, notamment des options pour la configuration du projet, les conventions de dénomination et les pratiques exemplaires. Cette page inclut également un exemple de fonction Lambda Node.js qui collecte des informations relatives à une commande, produit un reçu sous forme de fichier texte et place ce fichier dans un compartiment Amazon Simple Storage Service (Amazon S3). Pour plus d'informations sur le déploiement de votre fonction après l'avoir écrite, consultez the section called "Déploiement d'archives de fichiers .zip" ou the section called "Déploiement d'images de conteneur".

Rubriques

- Configuration de votre projet de gestionnaire Node.js
- Exemple de fonction Lambda Node.js
- Modules CommonJS et ES
- Initialisation de Node.js
- Convention de nommage du gestionnaire
- Définition et accès à l'objet d'événement d'entrée
- Modèles de gestionnaire valides pour les fonctions Node.js
- Utilisation du SDK pour la JavaScript version 3 dans votre gestionnaire
- Accès aux variables d'environnement
- Utilisation de l'état global
- Pratiques exemplaires en matière de code pour les fonctions Lambda Node.js

Configuration de votre projet de gestionnaire Node.js

Il existe plusieurs façons d'initialiser un projet Node.js Lambda. Par exemple, vous pouvez créer un projet Node.js standard en utilisant `npm`, créer une application AWS SAM ou créer une application AWS CDK.

Pour créer le projet avec `npm` :

```
npm init
```

Cette commande initialise votre projet et génère un fichier `package.json` qui gère les métadonnées et dépendances de votre projet.

Le code de votre fonction se trouve dans un `.mjs` JavaScript fichier `.js` or. Dans l'exemple suivant, nous nommons ce fichier `index.mjs`, car il utilise un gestionnaire de modules ES. Lambda prend en charge à la fois le module ES et les gestionnaires CommonJS. Pour de plus amples informations, veuillez consulter Modules CommonJS et ES.

Un projet de fonction Lambda Node.js typique suit cette structure générale :

```
/project-root
  ### index.mjs – Contains main handler
  ### package.json – Project metadata and dependencies
  ### package-lock.json – Dependency lock file
  ### node_modules/ – Installed dependencies
```

Exemple de fonction Lambda Node.js

L'exemple de code de fonction Lambda suivant prend en compte les informations relatives à une commande, produit un reçu sous forme de fichier texte et place ce fichier dans un compartiment Amazon S3.

Example index.mjs, fonction Lambda

```
import { S3Client, PutObjectCommand } from '@aws-sdk/client-s3';

// Initialize the S3 client outside the handler for reuse
const s3Client = new S3Client();

/**
 * Lambda handler for processing orders and storing receipts in S3.
 * @param {Object} event - Input event containing order details
 * @param {string} event.order_id - The unique identifier for the order
 * @param {number} event.amount - The order amount
 * @param {string} event.item - The item purchased
 * @returns {Promise<string>} Success message
 */
export const handler = async(event) => {
    try {
```

```
        // Access environment variables
        const bucketName = process.env.RECEIPT_BUCKET;
        if (!bucketName) {
            throw new Error('RECEIPT_BUCKET environment variable is not set');
        }

        // Create the receipt content and key destination
        const receiptContent = `OrderID: ${event.order_id}\nAmount: $
${event.amount.toFixed(2)}\nItem: ${event.item}`;
        const key = `receipts/${event.order_id}.txt`;

        // Upload the receipt to S3
        await uploadReceiptToS3(bucketName, key, receiptContent);

        console.log(`Successfully processed order ${event.order_id} and stored receipt
 in S3 bucket ${bucketName}`);
        return 'Success';
    } catch (error) {
        console.error(`Failed to process order: ${error.message}`);
        throw error;
    }
};

/**
 * Helper function to upload receipt to S3
 * @param {string} bucketName - The S3 bucket name
 * @param {string} key - The S3 object key
 * @param {string} receiptContent - The content to upload
 * @returns {Promise<void>}
 */
async function uploadReceiptToS3(bucketName, key, receiptContent) {
    try {
        const command = new PutObjectCommand({
            Bucket: bucketName,
            Key: key,
            Body: receiptContent
        });

        await s3Client.send(command);
    } catch (error) {
        throw new Error(`Failed to upload receipt to S3: ${error.message}`);
    }
}
```

Ce fichier `index.mjs` comprend les sections suivantes :

- Bloc `import` : utilisez ce bloc pour inclure les bibliothèques que votre fonction Lambda requiert, comme les clients du SDK AWS.
- Déclaration `const s3Client` : initialise un client Amazon S3 en dehors de la fonction du gestionnaire. Lambda exécute donc ce code pendant la phase d'initialisation, et le client est conservé pour être réutilisé lors de plusieurs appels.
- JSDoc bloc de commentaires : définissez les types d'entrée et de sortie pour votre gestionnaire à l'aide d'JSDoc annotations.
- `export const handler` : le gestionnaire de la fonction principal que Lambda invoque. Lorsque vous déployez votre fonction, spécifiez `index.handler` pour la propriété Gestionnaire. La valeur de la propriété `Handler` est le nom du fichier et le nom de la méthode du gestionnaire exporté, séparés par un point.
- Fonction `uploadReceiptToS3` : une fonction d'assistance référencée par la fonction du gestionnaire principal.

Pour que cette fonction fonctionne correctement, son rôle d'exécution doit autoriser l'action `s3:PutObject`. Assurez-vous également de définir la variable d'environnement `RECEIPT_BUCKET`. Après une invocation réussie, le compartiment Amazon S3 doit contenir un fichier de reçu.

Modules CommonJS et ES

Node.js prend en charge deux systèmes de modules : CommonJS et ECMAScript modules (modules ES). Lambda recommande d'utiliser les modules ES car ils prennent en charge l'attente de haut niveau, ce qui permet d'effectuer des tâches asynchrones lors de l'initialisation de l'environnement d'exécution.

Node.js traite les fichiers avec une extension de nom de `.cjs` fichier comme des modules CommonJS tandis qu'une `.mjs` extension désigne les modules ES. Par défaut, Node.js traite les fichiers portant l'extension de nom de `.js` fichier comme des modules CommonJS. Vous pouvez configurer Node.js pour traiter les `.js` fichiers comme des modules ES en spécifiant le `type` as `module` dans le `package.json` fichier de la fonction. Vous pouvez configurer Node.js dans Lambda pour détecter automatiquement si un `.js` fichier doit être traité comme CommonJS ou comme un module ES en ajoutant l'`–experimental-detect-module`indicateur à la `NODE_OPTIONS` variable d'environnement. Pour plus d'informations, consultez la section Fonctionnalités expérimentales de Node.js.

Les exemples suivants montrent des gestionnaires de fonctions écrits à l'aide de modules ES et de modules CommonJS. Les autres exemples de cette page utilisent tous des modules ES.

ES module example

Example – Gestionnaire de modules ES

```
const url = "https://aws.amazon.com/";

export const handler = async(event) => {
    try {
        const res = await fetch(url);
        console.info("status", res.status);
        return res.status;
    }
    catch (e) {
        console.error(e);
        return 500;
    }
};
```

CommonJS module example

Example – Gestionnaire de modules CommonJS

```
const https = require("https");
let url = "https://aws.amazon.com/";

exports.handler = async function (event) {
  let statusCode;
  await new Promise(function (resolve, reject) {
    https.get(url, (res) => {
        statusCode = res.statusCode;
        resolve(statusCode);
      }).on("error", (e) => {
        reject(Error(e));
      });
  });
  console.log(statusCode);
  return statusCode;
};
```

Initialisation de Node.js

Node.js utilise un I/O modèle non bloquant qui prend en charge des opérations asynchrones efficaces à l'aide d'une boucle d'événements. Par exemple, si Node.js effectue un appel réseau, la fonction continue de traiter les autres opérations sans bloquer une réponse réseau. Lorsque la réponse du réseau est reçue, elle est placée dans la file de rappel. Les tâches de la file d'attente sont traitées lorsque la tâche en cours est terminée.

Lambda recommande d'utiliser l'attente de haut niveau afin que les tâches asynchrones initiées lors de l'initialisation de l'environnement d'exécution soient terminées pendant l'initialisation. Les tâches asynchrones qui ne sont pas terminées lors de l'initialisation s'exécutent généralement lors du premier appel de fonction. Cela peut entraîner un comportement inattendu ou des erreurs. Par exemple, l'initialisation de votre fonction peut effectuer un appel réseau pour récupérer un paramètre depuis le Parameter Store AWS . Si cette tâche n'est pas terminée lors de l'initialisation, la valeur peut être nulle lors d'un appel. Il peut également y avoir un délai entre l'initialisation et l'appel, ce qui peut déclencher des erreurs lors d'opérations sensibles au facteur temps. En particulier, les appels de AWS service peuvent s'appuyer sur des signatures de demande urgentes, ce qui entraîne l'échec des appels de service si l'appel n'est pas terminé pendant la phase d'initialisation. L'exécution des tâches lors de l'initialisation améliore généralement les performances de démarrage à froid et les performances d'appel d'abord lors de l'utilisation de la concurrence provisionnée. Pour plus d'informations, consultez notre article de blog Using Node.js ES modules and top-level await in AWS Lambda.

Convention de nommage du gestionnaire

Lorsque vous configurez une fonction, la valeur du paramètre Gestionnaire est le nom du fichier et le nom de la méthode du gestionnaire exporté, séparés par un point. La valeur par défaut des fonctions créées dans la console et dans les exemples de ce guide est `index.handler`. Cela indique la méthode `handler` qui est exportée à partir du fichier `index.js` ou `index.mjs`.

Si vous créez une fonction dans la console en utilisant un nom de fichier ou un nom de gestionnaire de fonction différent, vous devez modifier le nom du gestionnaire par défaut.

Pour modifier le nom du gestionnaire de fonction (console)

1. Ouvrez la page Fonctions de la console Lambda et choisissez votre fonction.
2. Cliquez sur l’onglet Code.
3. Faites défiler l’écran jusqu’au volet Paramètres d’exécution et choisissez Modifier.

4. Dans Gestionnaire, saisissez le nouveau nom de votre gestionnaire de fonction.
5. Choisissez Enregistrer.

Définition et accès à l'objet d'événement d'entrée

JSON est le format d'entrée le plus courant et standard pour les fonctions Lambda. Dans cet exemple, la fonction exige une entrée similaire à l'exemple suivant :

```
{
    "order_id": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

Lorsque vous utilisez des fonctions Lambda dans Node.js, vous pouvez définir la forme attendue de l'événement d'entrée à l'aide JSDoc d'annotations. Dans cet exemple, nous définissons la structure de saisie dans le JSDoc commentaire du gestionnaire :

```
/**
 * Lambda handler for processing orders and storing receipts in S3.
 * @param {Object} event - Input event containing order details
 * @param {string} event.order_id - The unique identifier for the order
 * @param {number} event.amount - The order amount
 * @param {string} event.item - The item purchased
 * @returns {Promise<string>} Success message
 */
```

Après avoir défini ces types dans votre JSDoc commentaire, vous pouvez accéder aux champs de l'objet d'événement directement dans votre code. Par exemple, `event.order_id` récupère la valeur de `order_id` à partir de l'entrée d'origine.

Modèles de gestionnaire valides pour les fonctions Node.js

Nous vous recommandons d'utiliser async/await pour déclarer le gestionnaire de fonctions au lieu d'utiliser des rappels. Async/await est un moyen concis et lisible d'écrire du code asynchrone, sans avoir besoin de rappels imbriqués ou de promesses enchaînées. Avec async/await, vous pouvez écrire du code qui se lit comme du code synchrone, tout en étant asynchrone et non bloquant.

gestionnaires de fonctions asynchrones (recommandé)

Le mot-clé `async` marque une fonction comme étant asynchrone, et le mot-clé `await` met en pause l'exécution de la fonction jusqu'à ce qu'une `Promise` soit résolue. Le gestionnaire accepte les arguments suivants :

- `event` : contient les données d'entrée transmises à votre fonction.
- `context` : contient des informations sur l'invocation, la fonction et l'environnement d'exécution. Pour de plus amples informations, veuillez consulter Utilisation de l'objet de contexte Lambda pour récupérer les informations de la fonction Node.js.

Voici les signatures valides pour le async/await modèle :

```
export const handler = async (event) => { };
```

```
export const handler = async (event, context) => { };
```

Gestionnaires de fonctions synchrones

Lorsque votre fonction n'exécute aucune tâche asynchrone, vous pouvez utiliser un gestionnaire de fonctions synchrones en utilisant l'une des signatures de fonction suivantes :

```
export const handler = (event) => { };
```

```
export const handler = (event, context) => { };
```

Gestionnaires de fonctions de diffusion des réponses

Lambda prend en charge le streaming des réponses avec Node.js. Les gestionnaires de fonctions de streaming de réponses utilisent le `awslambda.streamifyResponse()` décorateur et prennent 3 paramètres : `eventresponseStream`, et. `context` La signature de la fonction est la suivante :

```
export const handler = awslambda.streamifyResponse(async (event, responseStream,
 context) => { });
```

Pour de plus amples informations, veuillez consulter Streaming de réponses pour les fonctions Lambda.

Gestionnaires de fonctions basés sur le rappel

> **Note**
>
> Les gestionnaires de fonctions basés sur le rappel ne sont pris en charge que jusqu'à Node.js 22. À partir de Node.js 24, les tâches asynchrones doivent être implémentées à l'aide de gestionnaires de fonctions asynchrones.

Les gestionnaires de fonctions basés sur le rappel doivent utiliser les arguments d'événement, de contexte et de rappel. Exemple :

```
export const handler = (event, context, callback) => { };
```

La fonction de rappel attend une `Error` et une réponse, qui doit être sérialisable en JSON. La fonction continue à s'exécuter jusqu'à ce que la boucle d'événement soit vide ou que la fonction s'arrête. La réponse n'est pas envoyée à l'appelant tant que toutes les tâches d'événement de boucle ne sont pas terminées. Si la fonction expire, une erreur est renvoyée à la place. Vous pouvez configurer le moteur d'exécution pour envoyer la réponse immédiatement en définissant le contexte. callbackWaitsForEmptyEventLoopà faux.

Example – Requête HTTP avec callback

L'exemple suivant vérifie la fonction d'une URL et renvoie le code de statut au mécanisme d'appel.

```
import https from "https";
let url = "https://aws.amazon.com/";

export const handler = (event, context, callback) => {
  https.get(url, (res) => {
    callback(null, res.statusCode);
  }).on("error", (e) => {
    callback(Error(e));
  });
};
```

Utilisation du SDK pour la JavaScript version 3 dans votre gestionnaire

Vous utiliserez souvent les fonctions Lambda pour interagir avec d'autres AWS ressources ou pour les mettre à jour. Le moyen le plus simple d'interagir avec ces ressources est d'utiliser AWS SDK

pour JavaScript. Tous les environnements d'exécution Lambda Node.js pris en charge incluent le SDK pour la version 3. JavaScript Toutefois, nous vous recommandons vivement d'inclure les clients AWS SDK dont vous avez besoin dans votre package de déploiement. Cela permet d'optimiser la rétrocompatibilité lors des futures mises à jour de l'environnement d'exécution Lambda. Ne vous fiez au SDK fourni par l'environnement d'exécution que lorsque vous ne pouvez pas inclure de packages supplémentaires (par exemple, lorsque vous utilisez l'éditeur de code de la console Lambda ou du code en ligne dans un modèle). AWS CloudFormation

Pour ajouter des dépendances au SDK à votre fonction, utilisez la commande `npm install` correspondant aux clients SDK spécifiques dont vous avez besoin. Dans l'exemple de code, nous avons utilisé le client Amazon S3. Ajoutez cette dépendance en exécutant la commande suivante dans le répertoire qui contient votre fichier `package.json` :

```
npm install @aws-sdk/client-s3
```

Dans le code de fonction, importez le client et les commandes dont vous avez besoin, comme le montre l'exemple de fonction :

```
import { S3Client, PutObjectCommand } from '@aws-sdk/client-s3';
```

Initialisez ensuite un client Amazon S3 :

```
const s3Client = new S3Client();
```

Dans cet exemple, nous avons initialisé notre client Amazon S3 en dehors de la fonction de gestionnaire principale pour éviter d'avoir à l'initialiser à chaque fois que nous invoquons notre fonction. Après avoir initialisé votre client SDK, vous pouvez l'utiliser pour effectuer des appels d'API pour ce AWS service. L'exemple de code appelle l'action d'PutObjectAPI Amazon S3 comme suit :

```
const command = new PutObjectCommand({
    Bucket: bucketName,
    Key: key,
    Body: receiptContent
});
```

Accès aux variables d'environnement

Dans le code de votre gestionnaire, vous pouvez référencer n'importe quelle variable d'environnement à l'aide de `process.env`. Dans cet exemple, nous référençons la variable d'environnement `RECEIPT_BUCKET` définie à l'aide des lignes de code suivantes :

```
// Access environment variables
const bucketName = process.env.RECEIPT_BUCKET;
if (!bucketName) {
    throw new Error('RECEIPT_BUCKET environment variable is not set');
}
```

Utilisation de l'état global

Lambda exécute votre code statique pendant la phase d'initialisation avant d'invoquer votre fonction pour la première fois. Les ressources créées pendant l'initialisation restent en mémoire entre les invocations, ce qui vous évite d'avoir à les créer chaque fois que vous invoquez votre fonction.

Dans l'exemple de code, le code d'initialisation du client S3 se trouve en dehors du gestionnaire. L'exécution initialise le client avant que la fonction n'exécute son premier événement, et le client reste disponible pour être réutilisé pendant toutes les invocations.

Pratiques exemplaires en matière de code pour les fonctions Lambda Node.js

Suivez ces instructions pour créer des fonctions Lambda :

- Séparez le gestionnaire Lambda de votre logique principale. Cela vous permet de créer une fonction testable plus unitaire.
- Contrôlez les dépendances du package de déploiement de vos fonctions. L'environnement AWS Lambda d'exécution contient un certain nombre de bibliothèques. Pour les environnements d'exécution Node.js et Python, ceux-ci incluent le AWS SDKs. Pour activer le dernier ensemble de mises à jour des fonctionnalités et de la sécurité, Lambda met régulièrement à jour ces bibliothèques. Ces mises à jour peuvent introduire de subtiles modifications dans le comportement de votre fonction Lambda. Pour disposer du contrôle total des dépendances que votre fonction utilise, empaquetez toutes vos dépendances avec votre package de déploiement.
- Réduisez la complexité de vos dépendances. Privilégiez les infrastructures plus simples qui se chargent rapidement au démarrage de l'environnement d'exécution.

- Réduisez la taille de votre package de déploiement selon ses besoins d'exécution. Cela contribue à réduire le temps nécessaire au téléchargement et à la décompression de votre package de déploiement avant l'invocation.

Tirez parti de la réutilisation de l'environnement d'exécution pour améliorer les performances de votre fonction. Initialisez les clients SDK et les connexions à la base de données en dehors du gestionnaire de fonctions et mettez en cache les actifs statiques localement dans le répertoire `/tmp`. Les invocations ultérieures traitées par la même instance de votre fonction peuvent réutiliser ces ressources. Cela permet d'économiser des coûts, tout en réduisant le temps d'exécution de la fonction.

Pour éviter des éventuelles fuites de données entre les invocations, n'utilisez pas l'environnement d'exécution pour stocker des données utilisateur, des événements ou d'autres informations ayant un impact sur la sécurité. Si votre fonction repose sur un état réversible qui ne peut pas être stocké en mémoire dans le gestionnaire, envisagez de créer une fonction distincte ou des versions distinctes d'une fonction pour chaque utilisateur.

Utilisez une directive keep-alive pour maintenir les connexions persistantes. Lambda purge les connexions inactives au fil du temps. Si vous tentez de réutiliser une connexion inactive lorsque vous invoquez une fonction, cela entraîne une erreur de connexion. Pour maintenir votre connexion persistante, utilisez la directive Keep-alive associée à votre environnement d'exécution. Pour obtenir un exemple, consultez Réutilisation des connexions avec Keep-Alive dans Node.js.

Utilisez des variables d'environnement pour transmettre des paramètres opérationnels à votre fonction. Par exemple, si vous écrivez dans un compartiment Amazon S3 au lieu de coder en dur le nom du compartiment dans lequel vous écrivez, configurez le nom du compartiment comme variable d'environnement.

Évitez d'utiliser des invocations récursives dans votre fonction Lambda, lorsque la fonction s'invoque elle-même ou démarre un processus susceptible de l'invoquer à nouveau. Cela peut entraîner un volume involontaire d'invocations de fonction et des coûts accrus. Si vous constatez un volume involontaire d'invocations, définissez immédiatement la simultanéité réservée à la fonction sur `0` afin de limiter toutes les invocations de la fonction, pendant que vous mettez à jour le code.

N'utilisez pas de code non documenté ni public APIs dans votre code de fonction Lambda. Pour les AWS Lambda environnements d'exécution gérés, Lambda applique régulièrement des mises à jour de sécurité et fonctionnelles aux applications internes de Lambda. APIs Ces mises à jour internes de l'API peuvent être rétroincompatibles, ce qui peut entraîner des conséquences imprévues, telles que

des échecs d'invocation si votre fonction dépend de ces mises à jour non publiques. APIs Consultez la référence de l'API pour obtenir une liste des API accessibles au public APIs.

Écriture du code idempotent. L'écriture de code idempotent pour vos fonctions garantit ne gestion identique des événements dupliqués. Votre code doit valider correctement les événements et gérer correctement les événements dupliqués. Pour de plus amples informations, veuillez consulterComment faire en sorte que ma fonction Lambda soit idempotente ?.

Déployer des fonctions Lambda en Node.js avec des archives de fichiers .zip

Le code de votre AWS Lambda fonction comprend un fichier .js ou .mjs contenant le code du gestionnaire de votre fonction, ainsi que les packages et modules supplémentaires dont dépend votre code. Pour déployer ce code de fonction vers Lambda, vous utilisez un package de déploiement. Ce package peut être une archive de fichier .zip ou une image de conteneur. Pour plus d'informations sur l'utilisation d'images de conteneur avec Node.js, consultez la page Déployer des fonctions Lambda Node.js avec des images de conteneur.

Pour créer votre package de déploiement sous forme d'archive de fichier .zip, vous pouvez utiliser l'utilitaire d'archivage .zip intégré à votre outil de ligne de commande, ou tout autre utilitaire .zip tel que 7zip. Les exemples présentés dans les sections suivantes supposent que vous utilisez un outil `zip` de ligne de commande dans un environnement Linux ou macOS. Pour utiliser les mêmes commandes sous Windows, vous pouvez installer le sous-système Windows pour Linux afin d'obtenir une version intégrée à Windows d'Ubuntu et de Bash.

Notez que Lambda utilise les autorisations de fichiers POSIX. Ainsi, vous pourriez devoir définir des autorisations pour le dossier du package de déploiement avant de créer l'archive de fichiers .zip.

Rubriques

- Dépendances d'exécution dans Node.js
- Création d'un package de déploiement .zip sans dépendances
- Création d'un package de déploiement .zip avec dépendances
- Création d'une couche Node.js pour vos dépendances
- Chemin de recherche des dépendances et bibliothèques incluses dans l'exécution
- Création et mise à jour de fonctions Lambda Node.js à l'aide de fichiers .zip

Dépendances d'exécution dans Node.js

Pour les fonctions Lambda qui utilisent l'exécution Node.js, une dépendance peut être n'importe quel package ou module Node.js. L'exécution Node.js comprend un certain nombre de bibliothèques courantes, ainsi qu'une version de AWS SDK pour JavaScript. Tous les environnements d'exécution Node.js pris en charge incluent la version 3 du SDK. Pour utiliser la version 2 du SDK, vous devez ajouter le SDK à votre package de déploiement de fichiers .zip. Pour savoir quelle version spécifique

du kit SDK est incluse dans l'environnement d'exécution que vous utilisez, consultez the section called "Versions du SDK incluses dans l'environnement d'exécution".

Lambda met régulièrement à jour les bibliothèques du kit SDK dans l'exécution Node.js afin d'inclure les dernières fonctionnalités et mises à jour de sécurité. Lambda applique également des correctifs de sécurité et des mises à jour aux autres bibliothèques incluses dans l'exécution. Pour avoir un contrôle total sur les dépendances de votre package, vous pouvez ajouter votre version préférée de n'importe quelle dépendance incluse dans l'exécution à votre package de déploiement. Par exemple, si vous souhaitez utiliser une version particulière du SDK pour JavaScript, vous pouvez l'inclure dans votre fichier .zip en tant que dépendance. Pour plus d'informations sur l'ajout de dépendances incluses dans l'exécution à votre fichier .zip, consultez Chemin de recherche des dépendances et bibliothèques incluses dans l'exécution.

Dans le cadre du modèle de responsabilité partagée AWS, vous êtes responsable de la gestion de toutes les dépendances dans les packages de déploiement de vos fonctions. Cela inclut l'application de mises à jour et de correctifs de sécurité. Pour mettre à jour les dépendances dans le package de déploiement de votre fonction, créez d'abord un nouveau fichier .zip, puis chargez-le sur Lambda. Pour plus d'informations, consultez Création d'un package de déploiement .zip avec dépendances et Création et mise à jour de fonctions Lambda Node.js à l'aide de fichiers .zip.

Création d'un package de déploiement .zip sans dépendances

Si le code de votre fonction ne comporte aucune dépendance à l'exception des bibliothèques incluses dans l'exécution Lambda, votre fichier .zip contient uniquement le fichier `index.js` ou `index.mjs` avec le code du gestionnaire de votre fonction. Utilisez votre utilitaire zip préféré pour créer un fichier .zip avec votre fichier `index.js` ou `index.mjs` à la racine. Si le fichier contenant le code de votre gestionnaire ne se trouve pas à la racine de votre fichier .zip, Lambda ne sera pas en mesure d'exécuter votre code.

Pour savoir comment déployer votre fichier .zip pour créer une nouvelle fonction Lambda ou mettre à jour une fonction Lambda existante, veuillez consulter la rubrique Création et mise à jour de fonctions Lambda Node.js à l'aide de fichiers .zip.

Création d'un package de déploiement .zip avec dépendances

Si votre code de fonction dépend de packages ou de modules qui ne sont pas inclus dans l'environnement d'exécution Lambda Node.js, vous pouvez soit ajouter ces dépendances à votre fichier .zip avec votre code de fonction, soit utiliser une couche Lambda. Les instructions de cette

section vous indiquent comment inclure vos dépendances dans votre package de déploiement .zip. Pour obtenir des instructions sur la façon d'inclure vos dépendances dans une couche, voir the section called "Création d'une couche Node.js pour vos dépendances".

L'exemple de commandes de CLI suivant crée un fichier .zip nommé `my_deployment_package.zip` contenant le fichier `index.js` ou `index.mjs` avec le gestionnaire de code de votre fonction et ses dépendances. Dans l'exemple, vous installez les dépendances à l'aide du gestionnaire de packages npm.

Pour créer le package de déploiement

1. Accédez au répertoire du projet qui contient votre fichier de code source `index.js` ou `index.mjs`. Dans cet exemple, le répertoire est nommé `my_function`.

   ```
   cd my_function
   ```

2. Installez les bibliothèques requises pour votre fonction dans le répertoire `node_modules` à l'aide de la commande `npm install`. Dans cet exemple, vous installez Kit SDK AWS X-Ray pour Node.js.

   ```
   npm install aws-xray-sdk
   ```

   Cela crée une structure de dossiers similaire à ce qui suit :

   ```
   ~/my_function
   ### index.mjs
   ### node_modules
       ### async
       ### async-listener
       ### atomic-batcher
       ### aws-sdk
       ### aws-xray-sdk
       ### aws-xray-sdk-core
   ```

   Vous pouvez également ajouter des modules personnalisés que vous créez vous-même à votre package de déploiement. Créez un répertoire sous `node_modules` portant le nom de votre module et enregistrez-y vos packages écrits personnalisés.

3. Créez un fichier .zip reprenant le contenu de votre dossier de projet à la racine. Utilisez l'option `r` (récursive) pour veiller à ce que zip compresse les sous-dossiers.

```
zip -r my_deployment_package.zip .
```

Création d'une couche Node.js pour vos dépendances

Les instructions de cette section vous indiquent comment inclure les dépendances dans une couche. Pour obtenir des instructions sur la façon d'inclure vos dépendances dans votre package de déploiement, voir the section called "Création d'un package de déploiement .zip avec dépendances".

Lorsque vous ajoutez une couche à une fonction, Lambda charge le contenu de la couche dans le répertoire `/opt` de cet environnement d'exécution. Pour chaque exécution Lambda, la variable `PATH` inclut déjà des chemins de dossiers spécifiques dans le répertoire `/opt`. Pour garantir que Lambda capture le contenu de votre couche, le fichier .zip de votre couche doit avoir ses dépendances un des chemins de dossier suivants :

- `nodejs/node_modules`
- `nodejs/node18/node_modules (NODE_PATH)`
- `nodejs/node20/node_modules (NODE_PATH)`
- `nodejs/node22/node_modules (NODE_PATH)`

Par exemple, la structure du fichier .zip de votre couche peut ressembler à ce qui suit :

```
xray-sdk.zip
# nodejs/node_modules/aws-xray-sdk
```

En outre, Lambda détecte automatiquement toutes les bibliothèques dans le répertoire `/opt/lib` et tous les fichiers binaires dans le répertoire `/opt/bin`. Pour que Lambda trouve correctement le contenu de votre couche, vous pouvez aussi créer une couche avec la structure suivante :

```
custom-layer.zip
# lib
    | lib_1
    | lib_2
# bin
    | bin_1
    | bin_2
```

Après avoir empaqueté votre couche, reportez-vous à the section called “Création et suppression de couches” et à the section called “Ajout de couches” pour terminer la configuration de votre couche.

Chemin de recherche des dépendances et bibliothèques incluses dans l’exécution

L’exécution Node.js comprend un certain nombre de bibliothèques courantes, ainsi qu’une version de AWS SDK pour JavaScript. Si vous souhaitez utiliser une version différente d’une bibliothèque incluse dans l’exécution, vous pouvez le faire en l’associant à votre fonction ou en l’ajoutant en tant que dépendance dans votre package de déploiement. Par exemple, vous pouvez utiliser une version différente du kit SDK en l’ajoutant à votre package de déploiement .zip. Vous pouvez également l’inclure dans une couche Lambda pour votre fonction.

Lorsque vous utilisez une instruction `import` ou `require` dans votre code, l’exécution Node.js recherche les répertoires dans le chemin `NODE_PATH` jusqu’à ce qu’elle trouve le module. Par défaut, le premier emplacement dans lequel recherche l’exécution est le répertoire dans lequel votre package de déploiement .zip est décompressé et monté (`/var/task`). Si vous ajoutez une version d’une bibliothèque incluse dans l’exécution dans votre package de déploiement, cette version prévaudra sur la version incluse dans l’exécution. Les dépendances de votre package de déploiement ont également la priorité sur les dépendances des couches.

Lorsque vous ajoutez une dépendance à une couche, Lambda l’extrait vers `/opt/nodejs/nodexx/node_modules`, où `nodexx` représente la version de l’exécution que vous utilisez. Dans le chemin de recherche, ce répertoire a la priorité sur le répertoire contenant les bibliothèques incluses dans l’exécution (`/var/lang/lib/node_modules`). Les bibliothèques des couches de fonctions ont donc la priorité sur les versions incluses dans l’exécution.

Vous pouvez voir le chemin de recherche complet de votre fonction Lambda en ajoutant la ligne de code suivante.

```
console.log(process.env.NODE_PATH)
```

Vous pouvez également ajouter des dépendances dans un dossier distinct au sein de votre package .zip. Par exemple, vous pouvez ajouter un module personnalisé à un dossier votre package .zip appelé `common`. Lorsque votre package .zip est décompressé et monté, ce dossier est placé dans le répertoire `/var/task`. Pour utiliser une dépendance provenant d’un dossier de votre package de déploiement .zip dans votre code, utilisez une instruction `import { } from` ou `const { } = require()`, selon que vous utilisez la résolution du module CJS ou ESM. Par exemple :

```
import { myModule } from './common'
```

Si vous regroupez votre code à `esbuild`, `rollup`, ou similaire, les dépendances utilisées par votre fonction sont regroupées dans un ou plusieurs fichiers. Nous vous recommandons d'utiliser cette méthode pour vendre des dépendances chaque fois que cela est possible. Par rapport à l'ajout de dépendances à votre package de déploiement, le regroupement de votre code améliore les performances grâce à la réduction I/O des opérations.

Création et mise à jour de fonctions Lambda Node.js à l'aide de fichiers .zip

Une fois que vous avez créé votre package de déploiement .zip, vous pouvez l'utiliser pour créer une nouvelle fonction Lambda ou mettre à jour une fonction Lambda existante. Vous pouvez déployer votre package .zip à l'aide de la console Lambda, de l'API Lambda et AWS Command Line Interface de l'API Lambda. Vous pouvez également créer et mettre à jour des fonctions Lambda à l'aide de l' AWS Serverless Application Model (AWS SAM) et de CloudFormation.

La taille maximale d'un package de déploiement .zip pour Lambda est de 250 Mo (décompressé). Notez que cette limite s'applique à la taille combinée de tous les fichiers que vous chargez, y compris les couches Lambda.

Le runtime Lambda a besoin d'une autorisation pour lire les fichiers de votre package de déploiement. Dans la notation octale des autorisations Linux, Lambda a besoin de 644 autorisations pour les fichiers non exécutables (rw-r--r--) et de 755 autorisations () pour les répertoires et les fichiers exécutables. rwxr-xr-x

Sous Linux et macOS, utilisez la commande `chmod` pour modifier les autorisations de fichiers sur les fichiers et les répertoires de votre package de déploiement. Par exemple, pour octroyer à un fichier non exécutable les autorisations correctes, exécutez la commande suivante.

```
chmod 644 <filepath>
```

Pour modifier les autorisations relatives aux fichiers dans Windows, voir Définir, afficher, modifier ou supprimer des autorisations sur un objet dans la documentation Microsoft Windows.

> **Note**
>
> Si vous n'accordez pas à Lambda les autorisations nécessaires pour accéder aux répertoires de votre package de déploiement, Lambda définit les autorisations pour ces répertoires sur 755 (). rwxr-xr-x

Création et mise à jour de fonctions avec des fichiers .zip à l'aide de la console

Pour créer une nouvelle fonction, vous devez d'abord créer la fonction dans la console, puis charger votre archive .zip. Pour mettre à jour une fonction existante, ouvrez la page de votre fonction, puis suivez la même procédure pour ajouter votre fichier .zip mis à jour.

Si votre fichier .zip fait moins de 50 Mo, vous pouvez créer ou mettre à jour une fonction en chargeant le fichier directement à partir de votre ordinateur local. Pour les fichiers .zip de plus de 50 Mo, vous devez d'abord charger votre package dans un compartiment Amazon S3. Pour savoir comment charger un fichier dans un compartiment Amazon S3 à l'aide du AWS Management Console, consultez Getting started with Amazon S3. Pour télécharger des fichiers à l'aide de AWS CLI, voir Déplacer des objets dans le guide de AWS CLI l'utilisateur.

> **Note**
>
> Vous ne pouvez pas modifier le type de package de déploiement (.zip ou image de conteneur) d'une fonction existante. Par exemple, vous ne pouvez pas convertir une fonction d'image de conteneur pour utiliser un fichier d'archive .zip à la place. Vous devez créer une nouvelle fonction.

Pour créer une nouvelle fonction (console)

1. Ouvrez la page Fonctions de la console Lambda et choisissez Créer une fonction.
2. Choisissez Créer à partir de zéro.
3. Sous Informations de base, procédez comme suit :
    a. Pour Nom de la fonction, saisissez le nom de la fonction.
    b. Pour Exécution, sélectionnez l'exécution que vous souhaitez utiliser.
    c. (Facultatif) Pour Architecture, choisissez l'architecture de l'ensemble des instructions pour votre fonction. L'architecture par défaut est x86_64. Assurez-vous que le package

de déploiement .zip pour votre fonction est compatible avec l'architecture de l'ensemble d'instructions que vous sélectionnez.

4. (Facultatif) Sous Permissions (Autorisations), développez Change default execution role (Modifier le rôle d'exécution par défaut). Vous pouvez créer un rôle d'exécution ou en utiliser un existant.
5. Choisissez Créer une fonction. Lambda crée une fonction de base « Hello world » à l'aide de l'exécution de votre choix.

Pour charger une archive .zip à partir de votre ordinateur local (console)

1. Sur la page Fonctions de la console Lambda, choisissez la fonction pour laquelle vous souhaitez charger le fichier .zip.
2. Sélectionnez l'onglet Code.
3. Dans le volet Source du code, choisissez Charger à partir de.
4. Choisissez Fichier .zip.
5. Pour charger un fichier .zip, procédez comme suit :
   a. Sélectionnez Charger, puis choisissez votre fichier .zip dans le sélecteur de fichiers.
   b. Choisissez Ouvrir.
   c. Choisissez Enregistrer.

Pour charger une archive .zip depuis un compartiment Amazon S3 (console)

1. Sur la page Fonctions de la console Lambda, choisissez la fonction pour laquelle vous souhaitez charger un nouveau fichier .zip.
2. Sélectionnez l'onglet Code.
3. Dans le volet Source du code, choisissez Charger à partir de.
4. Choisissez l'emplacement Amazon S3.
5. Collez l'URL du lien Amazon S3 de votre fichier .zip et choisissez Enregistrer.

Mise à jour des fonctions du fichier .zip à l'aide de l'éditeur de code de la console

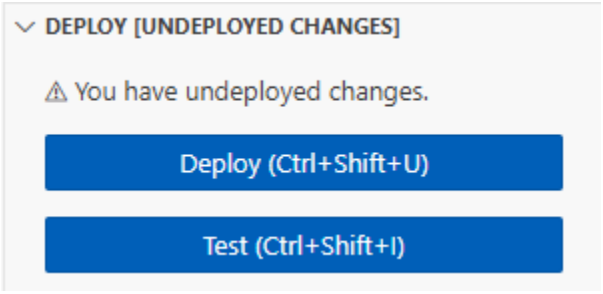
Pour certaines fonctions avec des packages de déploiement .zip, vous pouvez utiliser l'éditeur de code intégré de la console Lambda pour mettre à jour le code de votre fonction directement. Pour utiliser cette fonctionnalité, votre fonction doit répondre aux critères suivants :

- Votre fonction doit utiliser l'une des exécutions des langages interprétés (Python, Node.js ou Ruby).
- La taille du package de déploiement de votre fonction doit être inférieure à 50 Mo (décompressé).

Le code des fonctions avec les packages de déploiement d'images de conteneurs ne peut pas être édité directement dans la console.

Pour mettre à jour le code de fonction à l'aide de l'éditeur de code de la console

1. Ouvrez la page Fonctions de la console Lambda et choisissez votre fonction.
2. Sélectionnez l'onglet Code.
3. Dans le volet Source du code, sélectionnez votre fichier de code source et modifiez-le dans l'éditeur de code intégré.
4. Dans la section DÉPLOYER, choisissez Déployer pour mettre à jour le code de votre fonction :



Création et mise à jour de fonctions avec des fichiers .zip à l'aide du AWS CLI

Vous pouvez utiliser l'AWS CLI pour créer une nouvelle fonction ou pour mettre à jour une fonction existante à l'aide d'un fichier .zip. Utilisez la fonction de création et update-function-codeles commandes pour déployer votre package .zip. Si votre fichier .zip est inférieur à 50 Mo, vous pouvez charger le package .zip à partir d'un emplacement de fichier sur votre machine de génération locale. Pour les fichiers plus volumineux, vous devez charger votre package .zip à partir d'un compartiment Amazon S3. Pour savoir comment charger un fichier dans un compartiment Amazon S3 à l'aide du AWS CLI, consultez la section Déplacer des objets dans le guide de AWS CLI l'utilisateur.

> **Note**
>
> Si vous chargez votre fichier .zip depuis un compartiment Amazon S3 à l'aide de AWS CLI, le compartiment doit se trouver au même endroit Région AWS que votre fonction.

Pour créer une nouvelle fonction à l'aide d'un fichier .zip avec le AWS CLI, vous devez spécifier les éléments suivants :

- Le nom de votre fonction (`--function-name`)
- L'exécution de votre fonction (`--runtime`)
- L'Amazon Resource Name (ARN) du rôle d'exécution de votre fonction (`--role`)
- Le nom de la méthode du gestionnaire dans votre code de fonction (`--handler`)

Vous devez également indiquer l'emplacement de votre fichier .zip. Si votre fichier .zip se trouve dans un dossier sur votre machine de génération locale, utilisez l'option `--zip-file` pour spécifier le chemin d'accès du fichier, comme le montre l'exemple de commande suivant.

```
aws lambda create-function --function-name myFunction \
--runtime nodejs24.x --handler index.handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

Pour spécifier l'emplacement du fichier .zip dans un compartiment Amazon S3, utilisez l'option `--code` comme le montre l'exemple de commande suivant. Vous devez uniquement utiliser le paramètre `S3ObjectVersion` pour les objets soumis à la gestion des versions.

```
aws lambda create-function --function-name myFunction \
--runtime nodejs24.x --handler index.handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

Pour mettre à jour une fonction existante à l'aide de l'interface de ligne de commande, vous devez spécifier le nom de votre fonction à l'aide du paramètre `--function-name`. Vous devez également spécifier l'emplacement du fichier .zip que vous souhaitez utiliser pour mettre à jour votre code de fonction. Si votre fichier .zip se trouve dans un dossier sur votre machine de génération locale,

utilisez l'option `--zip-file` pour spécifier le chemin d'accès du fichier, comme le montre l'exemple de commande suivant.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Pour spécifier l'emplacement du fichier .zip dans un compartiment Amazon S3, utilisez les options `--s3-bucket` et `--s3-key` comme le montre l'exemple de commande suivant. Vous devez uniquement utiliser le paramètre `--s3-object-version` pour les objets soumis à la gestion des versions.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Création et mise à jour de fonctions avec des fichiers .zip à l'aide de l'API Lambda

Pour créer et mettre à jour des fonctions à l'aide d'une archive de fichiers .zip, utilisez les opérations d'API suivantes :

- CreateFunction
- UpdateFunctionCode

Création et mise à jour de fonctions avec des fichiers .zip à l'aide de AWS SAM

The AWS Serverless Application Model (AWS SAM) est une boîte à outils qui permet de rationaliser le processus de création et d'exécution d'applications sans serveur sur AWS. Vous définissez les ressources de votre application dans un modèle YAML ou JSON et vous utilisez l'interface de ligne de AWS SAM commande (AWS SAM CLI) pour créer, empaqueter et déployer vos applications. Lorsque vous créez une fonction Lambda à partir d'un AWS SAM modèle, elle crée AWS SAM automatiquement un package de déploiement ou une image de conteneur .zip avec le code de votre fonction et les dépendances que vous spécifiez. Pour en savoir plus sur l'utilisation des fonctions Lambda AWS SAM pour créer et déployer des fonctions Lambda, consultez Getting started with AWS SAM dans le Guide du AWS Serverless Application Model développeur.

Vous pouvez également l'utiliser AWS SAM pour créer une fonction Lambda à l'aide d'une archive de fichiers .zip existante. Pour créer une fonction Lambda à l'aide de AWS SAM, vous pouvez enregistrer votre fichier .zip dans un compartiment Amazon S3 ou dans un dossier local sur votre

machine de génération. Pour savoir comment charger un fichier dans un compartiment Amazon S3 à l'aide du AWS CLI, consultez la section Déplacer des objets dans le guide de AWS CLI l'utilisateur.

Dans votre AWS SAM modèle, la `AWS::Serverless::Function` ressource spécifie votre fonction Lambda. Dans cette ressource, définissez les propriétés suivantes pour créer une fonction à l'aide d'une archive de fichiers .zip :

- `PackageType` : défini sur `Zip`
- `CodeUri`- défini sur l'URI Amazon S3, le chemin d'accès au dossier local ou à l'FunctionCodeobjet du code de fonction
- `Runtime` : défini sur votre exécution choisie

Ainsi AWS SAM, si votre fichier .zip est supérieur à 50 Mo, vous n'avez pas besoin de le télécharger au préalable dans un compartiment Amazon S3. AWS SAM peut télécharger des packages .zip jusqu'à la taille maximale autorisée de 250 Mo (décompressés) à partir d'un emplacement sur votre machine de compilation locale.

Pour en savoir plus sur le déploiement de fonctions à l'aide d'un fichier .zip dans AWS SAM, consultez AWS::Serverless::Functionle manuel du AWS SAM développeur.

Création et mise à jour de fonctions avec des fichiers .zip à l'aide de CloudFormation

Vous pouvez l'utiliser CloudFormation pour créer une fonction Lambda à l'aide d'une archive de fichiers .zip. Pour créer une fonction Lambda à partir d'un fichier .zip, vous devez d'abord charger votre fichier dans un compartiment Amazon S3. Pour savoir comment charger un fichier dans un compartiment Amazon S3 à l'aide du AWS CLI, consultez la section Déplacer des objets dans le guide de AWS CLI l'utilisateur.

Dans votre CloudFormation modèle, la `AWS::Lambda::Function` ressource spécifie votre fonction Lambda. Dans cette ressource, définissez les propriétés suivantes pour créer une fonction à l'aide d'une archive de fichiers .zip :

- `PackageType` : défini sur `Zip`
- `Code` : saisissez le nom du compartiment Amazon S3 et le nom du fichier .zip dans les champs `S3Bucket` et `S3Key`
- `Runtime` : défini sur votre exécution choisie

Le fichier .zip CloudFormation généré ne peut pas dépasser 4 Mo. Pour en savoir plus sur le déploiement de fonctions à l'aide d'un fichier .zip dans CloudFormation, consultez AWS::Lambda::Functionle Guide de l'CloudFormation utilisateur.

Déployer des fonctions Lambda en Node.js avec des images conteneurs

Il existe trois méthodes pour créer une image de conteneur pour une fonction Lambda Node.js :

- Utilisation d'une image AWS de base pour Node.js

  Les images de base AWS sont préchargées avec une exécution du langage, un client d'interface d'exécution pour gérer l'interaction entre Lambda et votre code de fonction, et un émulateur d'interface d'exécution pour les tests locaux.

- Utilisation d'une image de base AWS uniquement pour le système d'exploitation

  AWS Les images de base réservées au système d'exploitation contiennent une distribution Amazon Linux et l'émulateur d'interface d'exécution. Ces images sont couramment utilisées pour créer des images de conteneur pour les langages compilés, tels que Go et Rust, et pour une langue ou une version linguistique pour laquelle Lambda ne fournit pas d'image de base, comme Node.js 19. Vous pouvez également utiliser des images de base uniquement pour le système d'exploitation pour implémenter un environnement d'exécution personnalisé. Pour rendre l'image compatible avec Lambda, vous devez inclure le client d'interface d'exécution pour Node.js dans l'image.

- Utilisation d'une image non AWS basique

  Vous pouvez utiliser une autre image de base à partir d'un autre registre de conteneur, comme Alpine Linux ou Debian. Vous pouvez également utiliser une image personnalisée créée par votre organisation. Pour rendre l'image compatible avec Lambda, vous devez inclure le client d'interface d'exécution pour Node.js dans l'image.

> **Tip**
>
> Pour réduire le temps nécessaire à l'activation des fonctions du conteneur Lambda, consultez Utiliser des générations en plusieurs étapes (français non garanti) dans la documentation Docker. Pour créer des images de conteneur efficaces, suivez la section Bonnes pratiques pour l'écriture de Dockerfiles (français non garanti).

Cette page explique comment créer, tester et déployer des images de conteneur pour Lambda.

Rubriques

- AWS images de base pour Node.js
- Utilisation d'une image AWS de base pour Node.js
- Utilisation d'une autre image de base avec le client d'interface d'exécution

AWS images de base pour Node.js

AWS fournit les images de base suivantes pour Node.js :

Étiquettes	Environnement d'exécution	Système d'exploitation	Dockerfile	Obsolescence
24	Node.js 24	Amazon Linux	Dockerfile pour Node.js 24 sur GitHub	30 avril 2028
22	Node.js 22	Amazon Linux	Dockerfile pour Node.js 22 sur GitHub	30 avril 2027
20	Node.js 20	Amazon Linux	Dockerfile pour Node.js 2.0 sur GitHub	30 avril 2026

Référentiel Amazon ECR : gallery.ecr. aws/lambda/nodejs

Les images de base de Node.js 20 et versions ultérieures sont basées sur l'image de conteneur minimale Amazon Linux 2023. Les images de base antérieures utilisaient Amazon Linux 2. AL2Le 023 offre plusieurs avantages par rapport à Amazon Linux 2, notamment un encombrement de déploiement réduit et des versions mises à jour de bibliothèques telles que`glibc`.

AL2Les images basées sur le format 023 sont utilisées `microdnf` (en lien symbolique comme`dnf`) comme gestionnaire de packages au lieu de`yum`, qui est le gestionnaire de packages par défaut dans Amazon Linux 2. `microdnf`est une implémentation autonome de. `dnf` Pour obtenir la liste des packages inclus dans les images AL2 basées sur la version 023, reportez-vous aux colonnes Conteneur minimal de la section Comparaison des packages installés sur les images de conteneurs Amazon Linux 2023. Pour plus d'informations sur les différences entre AL2 023 et Amazon Linux 2, consultez Présentation du runtime Amazon Linux 2023 AWS Lambda sur le blog AWS Compute.

> **Note**
>
> Pour exécuter des images AL2 basées sur 023 localement, y compris avec AWS Serverless Application Model (AWS SAM), vous devez utiliser Docker version 20.10.10 ou ultérieure.

Utilisation d'une image AWS de base pour Node.js

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- AWS CLI version 2
- Docker (version minimale 25.0.0)
- Le plug-in buildx Docker.
- Node.js

Création d'une image à partir d'une image de base

Pour créer une image de conteneur à partir d'une image AWS de base pour Node.js

1. Créez un répertoire pour le projet, puis passez à ce répertoire.

   ```
   mkdir example
   cd example
   ```

2. Créez un nouveau projet Node.js avec `npm`. Appuyez sur `Enter` pour accepter les options par défaut fournies dans l'expérience interactive.

   ```
   npm init
   ```

3. Créez un nouveau fichier appelé `index.js`. Vous pouvez ajouter l'exemple de code de fonction suivant au fichier pour le tester, ou utiliser le vôtre.

   Example Gestionnaire CommonJS

   ```
   exports.handler = async (event) => {
       const response = {
   ```

```
        statusCode: 200,
        body: JSON.stringify('Hello from Lambda!'),
    };
    return response;
};
```

4. Si votre fonction dépend de bibliothèques autres que celles-ci AWS SDK pour JavaScript, utilisez npm pour les ajouter à votre package.
5. Créez un nouveau Dockerfile avec la configuration suivante :

   - Définir la propriété FROM sur l'URI de l'image de base.
   - Utilisez la commande COPY pour copier le code de la fonction et les dépendances de l'environnement d'exécution dans {LAMBDA_TASK_ROOT}, une variable d'environnement définie par Lambda.
   - Définir l'argument CMD pour le gestionnaire de la fonction Lambda.

   Notez que l'exemple de Dockerfile n'inclut pas d'instruction USER. Lorsque vous déployez une image de conteneur sur Lambda, Lambda définit automatiquement un utilisateur Linux par défaut disposant d'autorisations de moindre privilège. Ceci est différent du comportement standard de Docker qui est défini par défaut par l'utilisateur `root` lorsqu'aucune instruction USER n'est fournie.

   Example Dockerfile

```
FROM public.ecr.aws/lambda/nodejs:22

# Copy function code
COPY index.js ${LAMBDA_TASK_ROOT}

# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "index.handler" ]
```

6. Générez l'image Docker à l'aide de la commande docker build. L'exemple suivant nomme l'image `docker-image` et lui donne la balise `test`. Pour rendre votre image compatible avec Lambda, vous devez utiliser l'option `--provenance=false`.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> La commande spécifie l'option `--platform linux/amd64` pour garantir la compatibilité de votre conteneur avec l'environnement d'exécution Lambda, quelle que soit l'architecture de votre machine de génération. Si vous avez l'intention de créer une fonction Lambda à l'aide de l'architecture du jeu ARM64 d'instructions, veillez à modifier la commande pour utiliser l'`--platform linux/arm64`option à la place.

(Facultatif) Testez l'image localement

1. Démarrez votre image Docker à l'aide de la commande docker run. Dans cet exemple, `docker-image` est le nom de l'image et `test` est la balise.

   ```
   docker run --platform linux/amd64 -p 9000:8080 docker-image:test
   ```

   Cette commande exécute l'image en tant que conteneur et crée un point de terminaison local à `localhost:9000/2015-03-31/functions/function/invocations`.

   > **Note**
   >
   > Si vous avez créé l'image Docker pour l'architecture du jeu ARM64 d'instructions, veillez à utiliser l'`--platform linux/arm64`option au lieu de`--platform linux/amd64`.

2. À partir d'une nouvelle fenêtre de terminal, publiez un événement au point de terminaison local.

   Linux/macOS

   Sous Linux et macOS, exécutez la commande `curl` suivante :

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   Cette commande invoque la fonction avec un événement vide et renvoie une réponse. Si vous utilisez votre propre code de fonction plutôt que l'exemple de code de fonction, vous pouvez invoquer la fonction avec une charge utile JSON. Exemple :

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

Dans PowerShell, exécutez la `Invoke-WebRequest` commande suivante :

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Cette commande invoque la fonction avec un événement vide et renvoie une réponse. Si vous utilisez votre propre code de fonction plutôt que l'exemple de code de fonction, vous pouvez invoquer la fonction avec une charge utile JSON. Exemple :

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. Obtenez l'ID du conteneur.

```
docker ps
```

4. Utilisez la commande docker kill pour arrêter le conteneur. Dans cette commande, remplacez `3766c4ab331c` par l'ID du conteneur de l'étape précédente.

```
docker kill 3766c4ab331c
```

Déploiement de l'image

Pour charger l'image sur Amazon RIE et créer la fonction Lambda

1. Exécutez la get-login-passwordcommande pour authentifier la CLI Docker auprès de votre registre Amazon ECR.
   - Définissez la `--region` valeur à l' Région AWS endroit où vous souhaitez créer le référentiel Amazon ECR.
   - `111122223333`Remplacez-le par votre Compte AWS identifiant.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Créez un référentiel dans Amazon ECR à l'aide de la commande create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Le référentiel Amazon ECR doit être Région AWS identique à la fonction Lambda.

En cas de succès, vous obtenez une réponse comme celle-ci :

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Copiez le `repositoryUri` à partir de la sortie de l'étape précédente.
4. Exécutez la commande docker tag pour étiqueter votre image locale dans votre référentiel Amazon ECR en tant que dernière version. Dans cette commande :

- `docker-image:test` est le nom et la balise de votre image Docker. Il s'agit du nom et de la balise de l'image que vous avez spécifiés dans la commande `docker build`.
- Remplacez `<ECRrepositoryUri>` par l'`repositoryUri` que vous avez copié. Assurez-vous d'inclure `:latest` à la fin de l'URI.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Exemple :

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Exécutez la commande docker push pour déployer votre image locale dans le référentiel Amazon ECR. Assurez-vous d'inclure `:latest` à la fin de l'URI du référentiel.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Créez un rôle d'exécution pour la fonction, si vous n'en avez pas déjà un. Vous aurez besoin de l'Amazon Resource Name (ARN) du rôle à l'étape suivante.
7. Créez la fonction Lambda. Pour `ImageUri`, indiquez l'URI du référentiel mentionné précédemment. Assurez-vous d'inclure `:latest` à la fin de l'URI.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> Vous pouvez créer une fonction en utilisant une image d'un autre AWS compte, à condition que l'image se trouve dans la même région que la fonction Lambda. Pour de plus amples informations, veuillez consulter Autorisations entre comptes Amazon ECR.

8. Invoquer la fonction.

```
aws lambda invoke --function-name hello-world response.json
```

Vous devriez obtenir une réponse comme celle-ci :

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Pour voir la sortie de la fonction, consultez le fichier `response.json`.

Pour mettre à jour le code de fonction, vous devez créer à nouveau l'image, télécharger la nouvelle image dans le référentiel Amazon ECR, puis utiliser la update-function-codecommande pour déployer l'image sur la fonction Lambda.

Lambda résout l'étiquette d'image en hachage d'image spécifique. Cela signifie que si vous pointez la balise d'image qui a été utilisée pour déployer la fonction vers une nouvelle image dans Amazon ECR, Lambda ne met pas automatiquement à jour la fonction pour utiliser la nouvelle image.

Pour déployer la nouvelle image sur la même fonction Lambda, vous devez utiliser la update-function-codecommande, même si la balise d'image dans Amazon ECR reste la même. Dans l'exemple suivant, l'option `--publish` crée une version de la fonction à l'aide de l'image du conteneur mise à jour.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Utilisation d'une autre image de base avec le client d'interface d'exécution

Si vous utilisez une image de base uniquement pour le système d'exploitation ou une autre image de base, vous devez inclure le client d'interface d'exécution dans votre image. Le client d'interface d'exécution étend leAPI de runtime, qui gère l'interaction entre Lambda et votre code de fonction.

Installez le client d'interface d'exécution Node.js à l'aide du gestionnaire de packages npm :

```
npm install aws-lambda-ric
```

Vous pouvez également télécharger le client d'interface d'exécution Node.js depuis GitHub.

L'exemple suivant montre comment créer une image de conteneur pour Node.js à l'aide d'une image non AWS basique. L'exemple Dockerfile utilise une image de base `bookworm`. Le Dockerfile inclut le client d'interface d'exécution.

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- AWS CLI version 2
- Docker (version minimale 25.0.0)
- Le plug-in buildx Docker.
- Node.js

Création d'une image à partir d'une image de base alternative

Pour créer une image de conteneur à partir d'une image non AWS basique

1. Créez un répertoire pour le projet, puis passez à ce répertoire.

```
mkdir example
cd example
```

2. Créez un nouveau projet Node.js avec `npm`. Appuyez sur `Enter` pour accepter les options par défaut fournies dans l'expérience interactive.

```
npm init
```

3. Créez un nouveau fichier appelé `index.js`. Vous pouvez ajouter l'exemple de code de fonction suivant au fichier pour le tester, ou utiliser le vôtre.

   Example Gestionnaire CommonJS

```
exports.handler = async (event) => {
    const response = {
        statusCode: 200,
        body: JSON.stringify('Hello from Lambda!'),
    };
```

```
    return response;
};
```

4. Créez un nouveau fichier Docker. Le Dockerfile suivant utilise une image de base `bookworm` au lieu d'une image de base AWS. Le Dockerfile inclut le client d'interface d'exécution, ce qui rend l'image compatible avec Lambda. Le Dockerfile utilise une création à plusieurs étapes. La première étape crée une image de génération, qui est un environnement Node.js standard dans lequel les dépendances de la fonction sont installées. La deuxième étape crée une image plus fine qui inclut le code de la fonction et ses dépendances. Cela permet de réduire la taille de l'image finale.

   - Définissez la propriété `FROM` pour l'identifiant de l'image de base.
   - Utilisez la commande `COPY` pour copier le code de la fonction et les dépendances de l'exécution.
   - Définissez le `ENTRYPOINT` sur le module que vous souhaitez que le conteneur Docker exécute lorsqu'il démarre. Dans ce cas, le module est le client d'interface d'exécution.
   - Définir l'argument `CMD` pour le gestionnaire de la fonction Lambda.

   Notez que l'exemple de Dockerfile n'inclut pas d'instruction USER. Lorsque vous déployez une image de conteneur sur Lambda, Lambda définit automatiquement un utilisateur Linux par défaut disposant d'autorisations de moindre privilège. Ceci est différent du comportement standard de Docker qui est défini par défaut par l'utilisateur `root` lorsqu'aucune instruction `USER` n'est fournie.

   Example Dockerfile

```
# Define custom function directory
ARG FUNCTION_DIR="/function"

FROM node:20-bookworm as build-image

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Install build dependencies
RUN apt-get update && \
    apt-get install -y \
    g++ \
    make \
```

```
    cmake \
    unzip \
    libcurl4-openssl-dev

# Copy function code
RUN mkdir -p ${FUNCTION_DIR}
COPY . ${FUNCTION_DIR}

WORKDIR ${FUNCTION_DIR}

# Install Node.js dependencies
RUN npm install

# Install the runtime interface client
RUN npm install aws-lambda-ric

# Grab a fresh slim copy of the image to reduce the final size
FROM node:20-bookworm-slim

# Required for Node runtimes which use npm@8.6.0+ because
# by default npm writes logs under /home/.npm and Lambda fs is read-only
ENV NPM_CONFIG_CACHE=/tmp/.npm

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Set working directory to function root directory
WORKDIR ${FUNCTION_DIR}

# Copy in the built dependencies
COPY --from=build-image ${FUNCTION_DIR} ${FUNCTION_DIR}

# Set runtime interface client as default command for the container runtime
ENTRYPOINT ["/usr/local/bin/npx", "aws-lambda-ric"]
# Pass the name of the function handler as an argument to the runtime
CMD ["index.handler"]
```

5. Générez l'image Docker à l'aide de la commande docker build. L'exemple suivant nomme l'image `docker-image` et lui donne la balise `test`. Pour rendre votre image compatible avec Lambda, vous devez utiliser l'option `--provenance=false`.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> La commande spécifie l'option `--platform linux/amd64` pour garantir la compatibilité de votre conteneur avec l'environnement d'exécution Lambda, quelle que soit l'architecture de votre machine de génération. Si vous avez l'intention de créer une fonction Lambda à l'aide de l'architecture du jeu ARM64 d'instructions, veillez à modifier la commande pour utiliser l'`--platform linux/arm64`option à la place.

(Facultatif) Testez l'image localement

Utilisez l'émulateur d'interface d'exécution pour tester l'image localement. Vous pouvez intégrer l'émulateur dans votre image ou utiliser la procédure suivante pour l'installer sur votre machine locale.

Pour installer et exécuter l'émulateur d'interface d'exécution sur votre ordinateur local

1. Depuis le répertoire de votre projet, exécutez la commande suivante pour télécharger l'émulateur d'interface d'exécution (architecture x86-64) GitHub et l'installer sur votre machine locale.

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   Pour installer l'émulateur arm64, remplacez l'URL du GitHub référentiel dans la commande précédente par la suivante :

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

   PowerShell

   ```
   $dirPath = "$HOME\.aws-lambda-rie"
   if (-not (Test-Path $dirPath)) {
       New-Item -Path $dirPath -ItemType Directory
   }
   ```

```
$downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/
releases/latest/download/aws-lambda-rie"
$destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

Pour installer l'émulateur arm64, remplacez `$downloadLink` par ce qui suit :

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. Démarrez votre image Docker à l'aide de la commande docker run. Notez ce qui suit :

   - `docker-image` est le nom de l'image et `test` est la balise.
   - `/usr/local/bin/npx aws-lambda-ric index.handler` est le ENTRYPOINT suivi du CMD depuis votre Dockerfile.

   Linux/macOS

   ```
   docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
    9000:8080 \
       --entrypoint /aws-lambda/aws-lambda-rie \
       docker-image:test \
           /usr/local/bin/npx aws-lambda-ric index.handler
   ```

   PowerShell

   ```
   docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
    9000:8080 `
   --entrypoint /aws-lambda/aws-lambda-rie `
   docker-image:test `
       /usr/local/bin/npx aws-lambda-ric index.handler
   ```

   Cette commande exécute l'image en tant que conteneur et crée un point de terminaison local à `localhost:9000/2015-03-31/functions/function/invocations`.

> **Note**
>
> Si vous avez créé l'image Docker pour l'architecture du jeu ARM64 d'instructions, veillez à utiliser l'`--platform linux/arm64`option au lieu de`--platform linux/amd64`.

3. Publiez un événement au point de terminaison local.

   Linux/macOS

   Sous Linux et macOS, exécutez la commande `curl` suivante :

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   Cette commande invoque la fonction avec un événement vide et renvoie une réponse. Si vous utilisez votre propre code de fonction plutôt que l'exemple de code de fonction, vous pouvez invoquer la fonction avec une charge utile JSON. Exemple :

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   Dans PowerShell, exécutez la `Invoke-WebRequest` commande suivante :

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```

   Cette commande invoque la fonction avec un événement vide et renvoie une réponse. Si vous utilisez votre propre code de fonction plutôt que l'exemple de code de fonction, vous pouvez invoquer la fonction avec une charge utile JSON. Exemple :

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
    "application/json"
   ```

4. Obtenez l'ID du conteneur.

   ```
   docker ps
   ```

5. Utilisez la commande docker kill pour arrêter le conteneur. Dans cette commande, remplacez `3766c4ab331c` par l'ID du conteneur de l'étape précédente.

```
docker kill 3766c4ab331c
```

Déploiement de l'image

Pour charger l'image sur Amazon RIE et créer la fonction Lambda

1. Exécutez la get-login-passwordcommande pour authentifier la CLI Docker auprès de votre registre Amazon ECR.
   - Définissez la `--region` valeur à l' Région AWS endroit où vous souhaitez créer le référentiel Amazon ECR.
   - `111122223333`Remplacez-le par votre Compte AWS identifiant.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Créez un référentiel dans Amazon ECR à l'aide de la commande create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Le référentiel Amazon ECR doit être Région AWS identique à la fonction Lambda.

En cas de succès, vous obtenez une réponse comme celle-ci :

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
```

```
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Copiez le `repositoryUri` à partir de la sortie de l'étape précédente.
4. Exécutez la commande docker tag pour étiqueter votre image locale dans votre référentiel Amazon ECR en tant que dernière version. Dans cette commande :

   - `docker-image:test` est le nom et la balise de votre image Docker. Il s'agit du nom et de la balise de l'image que vous avez spécifiés dans la commande `docker build`.
   - Remplacez <ECRrepositoryUri> par l'`repositoryUri` que vous avez copié. Assurez-vous d'inclure `:latest` à la fin de l'URI.

   ```
   docker tag docker-image:test <ECRrepositoryUri>:latest
   ```

   Exemple :

   ```
   docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
   world:latest
   ```

5. Exécutez la commande docker push pour déployer votre image locale dans le référentiel Amazon ECR. Assurez-vous d'inclure `:latest` à la fin de l'URI du référentiel.

   ```
   docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

6. Créez un rôle d'exécution pour la fonction, si vous n'en avez pas déjà un. Vous aurez besoin de l'Amazon Resource Name (ARN) du rôle à l'étape suivante.
7. Créez la fonction Lambda. Pour `ImageUri`, indiquez l'URI du référentiel mentionné précédemment. Assurez-vous d'inclure `:latest` à la fin de l'URI.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> Vous pouvez créer une fonction en utilisant une image d'un autre AWS compte, à condition que l'image se trouve dans la même région que la fonction Lambda. Pour de plus amples informations, veuillez consulter Autorisations entre comptes Amazon ECR.

8. Invoquer la fonction.

```
aws lambda invoke --function-name hello-world response.json
```

Vous devriez obtenir une réponse comme celle-ci :

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Pour voir la sortie de la fonction, consultez le fichier `response.json`.

Pour mettre à jour le code de fonction, vous devez créer à nouveau l'image, télécharger la nouvelle image dans le référentiel Amazon ECR, puis utiliser la update-function-codecommande pour déployer l'image sur la fonction Lambda.

Lambda résout l'étiquette d'image en hachage d'image spécifique. Cela signifie que si vous pointez la balise d'image qui a été utilisée pour déployer la fonction vers une nouvelle image dans Amazon ECR, Lambda ne met pas automatiquement à jour la fonction pour utiliser la nouvelle image.

Pour déployer la nouvelle image sur la même fonction Lambda, vous devez utiliser la update-function-codecommande, même si la balise d'image dans Amazon ECR reste la même. Dans l'exemple suivant, l'option `--publish` crée une version de la fonction à l'aide de l'image du conteneur mise à jour.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Utilisation de couches pour les fonctions Lambda Node.js

Utilisez les couches Lambda pour empaqueter le code et les dépendances que vous souhaitez réutiliser dans plusieurs fonctions. Les couches contiennent généralement des dépendances de bibliothèque, une exécution personnalisée, ou des fichiers de configuration. La création d'une couche implique trois étapes générales :

1. Empaquetez le contenu de votre couche. Cela signifie créer une archive de fichiers .zip contenant les dépendances que vous souhaitez utiliser dans vos fonctions.
2. Créez la couche dans Lambda.
3. Ajoutez la couche à vos fonctions.

Rubriques

- Empaqueter le contenu de votre couche
- Créer la couche dans Lambda
- Ajouter la couche à votre fonction
- Exemple d'application

Empaqueter le contenu de votre couche

Pour créer une couche, regroupez vos packages dans une archive (fichier .zip) répondant aux exigences suivantes :

- Créez la couche à l'aide de la même version de Node.js que celle que vous prévoyez d'utiliser pour la fonction Lambda. Par exemple, si vous créez votre couche à l'aide de Node.js 24, utilisez le runtime Node.js 24 pour votre fonction.
- Le fichier .zip de votre couche doit utiliser l'une des structures de répertoires suivantes :
  - `nodejs/node_modules`
  - `nodejs/node`*X*`/node_modules`(où se *X* trouve votre version de Node.js, par exemple`node22`)

  Pour de plus amples informations, veuillez consulter Chemins d'accès de couche pour chaque exécution Lambda.
- Les packages de votre couche doivent être compatibles avec Linux. Les fonctions Lambda s'exécutent sur Amazon Linux.

Vous pouvez créer des couches contenant des bibliothèques Node.js tierces installées avec `npm` (telles que `axios` ou `lodash`) ou vos propres JavaScript modules.

Dépendances tierces

Créer une couche à l'aide de packages npm

1. Créez la structure de répertoires requise et installez-y les packages directement :

   ```
   mkdir -p nodejs
   npm install --prefix nodejs lodash axios
   ```

   Cette commande installe les packages directement dans le répertoire `nodejs/node_modules`, qui correspond à la structure requise par Lambda.

   > **Note**
   >
   > Pour les packages avec des dépendances natives ou des composants binaires (comme sharp ou bcrypt), assurez-vous qu'ils sont compatibles avec l'environnement Linux Lambda et l'architecture de votre fonction. Vous devrez peut-être utiliser l'indicateur `--platform` :
   >
   > ```
   > npm install --prefix nodejs --platform=linux --arch=x64 sharp
   > ```
   >
   > Pour les dépendances natives plus complexes, vous devrez peut-être les compiler dans un environnement Linux qui correspond à l'environnement d'exécution Lambda. Vous pouvez utiliser Docker à cette fin.

2. Compressez le contenu de la couche :

   Linux/macOS

   ```
   zip -r layer.zip nodejs/
   ```

   PowerShell

   ```
   Compress-Archive -Path .\nodejs -DestinationPath .\layer.zip
   ```

La structure de répertoire de votre fichier .zip devrait se présenter comme suit :

```
nodejs/
### package.json
### package-lock.json
### node_modules/
    ### lodash/
    ### axios/
    ### (dependencies of the other packages)
```

> **Note**
>
> - Assurez-vous que votre fichier .zip inclut le répertoire `nodejs` au niveau racine avec `node_modules` à l'intérieur. Cette structure garantit que Lambda peut localiser et importer vos packages.
> - Les fichiers `package.json` et `package-lock.json` du répertoire `nodejs/` sont utilisés par npm pour la gestion des dépendances, mais ne sont pas requis par Lambda pour le fonctionnement des couches. Chaque package installé contient déjà son propre fichier `package.json` qui définit la manière dont Lambda importe le package.

JavaScript Modules personnalisés

Créer une couche à l'aide de votre propre code

1. Créez la structure de répertoires requise pour votre couche :

```
mkdir -p nodejs/node_modules/validator
cd nodejs/node_modules/validator
```

2. Créez un fichier `package.json` pour votre module personnalisé afin de définir comment il doit être importé :

   Example nodejs/node_modules/validator/package.json

```
{
```

```
  "name": "validator",
  "version": "1.0.0",
  "type": "module",
  "main": "index.mjs"
}
```

3. Créez votre fichier JavaScript de module :

   Example nodejs/node_modules/validator/index.mjs

```
export function validateOrder(orderData) {
  // Validates an order and returns formatted data
  const requiredFields = ['productId', 'quantity'];

  // Check required fields
  const missingFields = requiredFields.filter(field => !(field in orderData));
  if (missingFields.length > 0) {
    throw new Error(`Missing required fields: ${missingFields.join(', ')}`);
  }

  // Validate quantity
  const quantity = orderData.quantity;
  if (!Number.isInteger(quantity) || quantity < 1) {
    throw new Error('Quantity must be a positive integer');
  }

  // Format and return the validated data
  return {
    productId: String(orderData.productId),
    quantity: quantity,
    shippingPriority: orderData.priority || 'standard'
  };
}

export function formatResponse(statusCode, body) {
  // Formats the API response
  return {
    statusCode: statusCode,
    body: JSON.stringify(body)
  };
}
```

4. Compressez le contenu de la couche :

Linux/macOS

```
zip -r layer.zip nodejs/
```

PowerShell

```
Compress-Archive -Path .\nodejs -DestinationPath .\layer.zip
```

La structure de répertoire de votre fichier .zip devrait se présenter comme suit :

```
nodejs/
### node_modules/
    ### validator/
        ### package.json
        ### index.mjs
```

5. Dans votre fonction, importez et utilisez les modules. Exemple :

```
import { validateOrder, formatResponse } from 'validator';

export const handler = async (event) => {
  try {
    // Parse the order data from the event body
    const orderData = JSON.parse(event.body || '{}');

    // Validate and format the order
    const validatedOrder = validateOrder(orderData);

    return formatResponse(200, {
      message: 'Order validated successfully',
      order: validatedOrder
    });
  } catch (error) {
    if (error instanceof Error && error.message.includes('Missing required
 fields')) {
      return formatResponse(400, {
        error: error.message
      });
    }
```

```
    return formatResponse(500, {
      error: 'Internal server error'
    });
  }
};
```

Vous pouvez utiliser l'événement de test suivant afin d'invoquer la fonction :

```
{
    "body": "{\"productId\": \"ABC123\", \"quantity\": 2, \"priority\": \"express
\"}"
}
```

Réponse attendue :

```
{
  "statusCode": 200,
  "body": "{\"message\":\"Order validated successfully\",\"order\":{\"productId\":
\"ABC123\",\"quantity\":2,\"shippingPriority\":\"express\"}}"
}
```

Créer la couche dans Lambda

Vous pouvez publier votre couche à l'aide de la console Lambda AWS CLI ou de la console Lambda.

AWS CLI

Exécutez la publish-layer-version AWS CLI commande pour créer la couche Lambda :

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip
 --compatible-runtimes nodejs24.x
```

Le paramètre compatible runtimes est facultatif. Lorsqu'il est spécifié, Lambda utilise ce paramètre pour filtrer les couches dans la console Lambda.

Console

Pour créer une couche (console)

1. Ouvrez la page Couches de la console Lambda.

2. Sélectionnez Créer un calque.
3. Choisissez Charger un fichier .zip, puis chargez l'archive .zip que vous avez créée précédemment.
4. (Facultatif) Pour Environnements d'exécution compatibles, choisissez l'environnement d'exécution Node.js qui correspond à la version de Node.js que vous avez utilisée pour créer votre couche.
5. Choisissez Créer.

Ajouter la couche à votre fonction

AWS CLI

Pour associer la couche à votre fonction, exécutez la update-function-configuration AWS CLI commande. Utilisez l'ARN de la couche pour le paramètre `--layers`. L'ARN doit spécifier la version (par exemple, `arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`). Pour de plus amples informations, veuillez consulter Couches et versions de couches.

```
aws lambda update-function-configuration --function-name my-function --cli-binary-format raw-in-base64-out --layers "arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1"
```

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

Console

Ajouter une couche à une fonction

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez la fonction.
3. Faites défiler jusqu'à la section Couches, puis choisissez Ajouter une couche.
4. Sous Choisir une couche, sélectionnez Couches personnalisées, puis choisissez votre couche.

> **Note**
>
> Si vous n'avez pas ajouté d'environnement d'exécution compatible lors de la création de la couche, celle-ci ne sera pas répertoriée ici. Vous pouvez spécifier l'ARN de la couche à la place.

5. Choisissez Ajouter.

Exemple d'application

Pour d'autres exemples d'utilisation des couches Lambda, consultez l'exemple d'application layer-nodejs dans le référentiel du Developer Guide. AWS Lambda GitHub Cette application inclut une couche qui contient la bibliothèque lodash. Après avoir créé la couche, vous pouvez déployer et invoquer la fonction correspondante pour confirmer que la couche fonctionne comme prévu.

Utilisation de l'objet de contexte Lambda pour récupérer les informations de la fonction Node.js

Lorsque Lambda exécute votre fonction, il transmet un objet contexte au gestionnaire. Cet objet fournit des méthodes et des propriétés fournissant des informations sur l'invocation, la fonction et l'environnement d'exécution.

Méthodes de contexte

- `getRemainingTimeInMillis()` – Renvoie le nombre de millisecondes restant avant l'expiration de l'exécution.

Propriétés du contexte

- `functionName` – Nom de la fonction Lambda.
- `functionVersion` – Version de la fonction.
- `invokedFunctionArn` – Amazon Resource Name (ARN) utilisé pour appeler la fonction. Indique si l'appelant a spécifié un numéro de version ou un alias.
- `memoryLimitInMB` – Quantité de mémoire allouée à la fonction.
- `awsRequestId` – Identifiant de la demande d'invocation.
- `logGroupName` – Groupe de journaux pour la fonction.
- `logStreamName` – Flux de journal de l'instance de fonction.
- `identity` – (applications mobiles) Informations sur l'identité Amazon Cognito qui a autorisé la demande.
  - `cognitoIdentityId` – Identité Amazon Cognito authentifiée.
  - `cognitoIdentityPoolId` – Groupe d'identités Amazon Cognito ayant autorisé l'invocation.
- `clientContext` – (applications mobiles) Contexte client fourni à Lambda par l'application client.
  - `client.installation_id`
  - `client.app_title`
  - `client.app_version_name`
  - `client.app_version_code`
  - `client.app_package_name`
  - `env.platform_version`

  - `env.platform`
  - `env.make`
  - `env.model`
  - `env.locale`
  - `custom` – Personnalisez les valeurs qui sont définies par l'application client.
- `callbackWaitsForEmptyEventLoop`— Par défaut (`true`), lors de l'utilisation d'un gestionnaire de fonctions basé sur le rappel, Lambda attend que la boucle d'événements soit vide après l'exécution du rappel avant de mettre fin à l'appel de la fonction. Définissez cette valeur `false` sur pour envoyer la réponse et mettre fin à l'appel immédiatement après l'exécution du rappel au lieu d'attendre que la boucle d'événements soit vide. Les événements exceptionnels continuent de se dérouler lors de la prochaine invocation. Notez que Lambda prend en charge les gestionnaires de fonctions basés sur le rappel pour les environnements d'exécution de Node.js 22 et antérieurs uniquement.

Dans l'exemple suivant, la fonction enregistre des informations de contexte et renvoie l'emplacement des journaux.

Example Fichier index.js

```
exports.handler = async function(event, context) {
  console.log('Remaining time: ', context.getRemainingTimeInMillis())
  console.log('Function name: ', context.functionName)
  return context.logStreamName
}
```

Journalisation et surveillance des fonctions Lambda Node.js

AWS Lambda surveille automatiquement les fonctions Lambda en votre nom et envoie les journaux à Amazon. CloudWatch Votre fonction Lambda est fournie avec un groupe de CloudWatch journaux Logs et un flux de journaux pour chaque instance de votre fonction. L'environnement d'exécution Lambda envoie des informations sur chaque invocation au flux de journaux et transmet les journaux et autres sorties provenant du code de votre fonction. Pour de plus amples informations, veuillez consulter Envoi automatique des journaux de la fonction Lambda à CloudWatch Logs.

Cette page explique comment générer une sortie de journal à partir du code de votre fonction Lambda et comment accéder aux journaux à l'aide de la AWS Command Line Interface console Lambda ou de la console. CloudWatch

Sections

- Création d'une fonction qui renvoie des journaux
- Utilisation des contrôles de journalisation avancés de Lambda avec Node.js
- Affichage des journaux dans la console Lambda
- Afficher les journaux dans la CloudWatch console
- Afficher les journaux à l'aide de AWS Command Line Interface (AWS CLI)
- Suppression de journaux

Création d'une fonction qui renvoie des journaux

Pour générer les journaux à partir de votre code de fonction, vous pouvez utiliser des méthodes sur l'objet console ou n'importe quelle bibliothèque de journalisation qui écrit dans `stdout` ou `stderr`. L'exemple suivant consigne les valeurs des variables d'environnement et l'objet d'événement.

> **Note**
>
> Nous vous recommandons d'utiliser des techniques telles que la validation des entrées et le codage des sorties lors de la journalisation des entrées. Si vous journalisez directement les données d'entrée, un attaquant peut utiliser votre code pour rendre les altérations difficiles à détecter, falsifier des entrées de journal ou contourner les moniteurs de journal. Pour plus d'informations, consultez Improper Output Neutralization for Logs dans la Common Weakness Enumeration.

Example Fichier index.js – Journalisation

```
exports.handler = async function(event, context) {
  console.log("ENVIRONMENT VARIABLES\n" + JSON.stringify(process.env, null, 2))
  console.info("EVENT\n" + JSON.stringify(event, null, 2))
  console.warn("Event not processed.")
  return context.logStreamName
}
```

Example format des journaux

```
START RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac Version: $LATEST
2019-06-07T19:11:20.562Z c793869b-ee49-115b-a5b6-4fd21e8dedac INFO ENVIRONMENT
 VARIABLES
{
  "AWS_LAMBDA_FUNCTION_VERSION": "$LATEST",
  "AWS_LAMBDA_LOG_GROUP_NAME": "/aws/lambda/my-function",
  "AWS_LAMBDA_LOG_STREAM_NAME": "2019/06/07/[$LATEST]e6f4a0c4241adcd70c262d34c0bbc85c",
  "AWS_EXECUTION_ENV": "AWS_Lambda_nodejs12.x",
  "AWS_LAMBDA_FUNCTION_NAME": "my-function",
  "PATH": "/var/lang/bin:/usr/local/bin:/usr/bin/:/bin:/opt/bin",
  "NODE_PATH": "/opt/nodejs/node10/node_modules:/opt/nodejs/node_modules:/var/runtime/
node_modules",
  ...
}
2019-06-07T19:11:20.563Z c793869b-ee49-115b-a5b6-4fd21e8dedac INFO EVENT
{
  "key": "value"
}
2019-06-07T19:11:20.564Z c793869b-ee49-115b-a5b6-4fd21e8dedac WARN Event not processed.
END RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac
REPORT RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac Duration: 128.83 ms Billed
 Duration: 296 ms Memory Size: 128 MB Max Memory Used: 74 MB Init Duration: 166.62 ms
 XRAY TraceId: 1-5d9d007f-0a8c7fd02xmpl480aed55ef0 SegmentId: 3d752xmpl1bbe37e Sampled:
 true
```

Le runtime Node.js enregistre les lignes `START`, `END` et `REPORT` pour chaque invocation. Il ajoute un horodatage, un ID de demande et un niveau de journalisation sur chaque entrée consignée par la fonction. La ligne de rapport fournit les détails suivants.

Champs de données de la ligne REPORT

- RequestId— L'identifiant de demande unique pour l'invocation.
- Duration – Temps que la méthode de gestion du gestionnaire de votre fonction a consacré au traitement de l'événement.
- Billed Duration : temps facturé pour l'invocation.
- Memory Size – Quantité de mémoire allouée à la fonction.
- Max Memory Used – Quantité de mémoire utilisée par la fonction. Lorsque les appels partagent un environnement d'exécution, Lambda indique la mémoire maximale utilisée pour toutes les invocations. Ce comportement peut entraîner une valeur signalée plus élevée que prévu.
- Init Duration : pour la première requête servie, temps qu'il a pris à l'exécution charger la fonction et exécuter le code en dehors de la méthode du gestionnaire.
- XRAY TraceId — Pour les demandes suivies, l'ID de AWS X-Ray trace.
- SegmentId— Pour les demandes tracées, l'identifiant du segment X-Ray.
- Sampled – Pour les demandes suivies, résultat de l'échantillonnage.

Vous pouvez consulter les journaux dans la console Lambda, dans la console CloudWatch Logs ou depuis la ligne de commande.

Utilisation des contrôles de journalisation avancés de Lambda avec Node.js

Pour mieux contrôler la manière dont les journaux de vos fonctions sont capturés, traités et consommés, vous pouvez configurer les options de journalisation suivantes pour les environnements d'exécution Node.js pris en charge :

- Format de journal : choisissez entre le format texte brut et le format JSON structuré pour les journaux de votre fonction
- Niveau de journalisation : pour les journaux au format JSON, choisissez le niveau de détail des journaux que Lambda envoie à Amazon CloudWatch, par exemple ERROR, DEBUG ou INFO
- Groupe de journaux : choisissez le groupe de CloudWatch journaux auquel votre fonction envoie les journaux

Pour plus d'informations sur ces options de journalisation et pour savoir comment configurer votre fonction pour les utiliser, consultez the section called "Configuration de commandes de journalisation avancées pour votre fonction Lambda".

Pour utiliser le format de journal et les options de niveau de journal avec vos fonctions Node.js Lambda, consultez les instructions des sections suivantes.

Utilisation de journaux JSON structurés avec Node.js

Si vous sélectionnez JSON pour le format de journal de votre fonction, Lambda enverra la sortie des journaux à l'aide des méthodes de consoleconsole.trace,,,console.debug, console.log console.infoconsole.error, et console.warn vers CloudWatch sous forme de JSON structuré. Chaque objet de journal JSON contient au moins quatre paires clé-valeur avec les clés suivantes :

- "timestamp" - heure à laquelle le message de journal a été généré
- "level" - niveau de journalisation attribué au message
- "message" - contenu du message de journal
- "requestId" - identifiant unique de la demande pour l'invocation de la fonction

Selon la méthode de journalisation utilisée par votre fonction, cet objet JSON peut également contenir des paires de clés supplémentaires. Par exemple, si votre fonction utilise des méthodes console pour enregistrer les objets d'erreur à l'aide de plusieurs arguments, l'objet JSON contiendra des paires clé-valeur supplémentaires avec les clés errorMessage, errorType et stackTrace.

Si votre code utilise déjà une autre bibliothèque de journalisation, telle que Powertools for AWS Lambda, pour produire des journaux structurés en JSON, vous n'avez pas besoin d'apporter de modifications. Lambda n'encode pas deux fois les journaux déjà codés en JSON. Les journaux d'application de votre fonction continueront donc d'être capturés comme avant.

Pour plus d'informations sur l'utilisation du package Powertools for AWS Lambda logging afin de créer des journaux structurés JSON dans le runtime Node.js, consultezthe section called "Logging".

Exemple de sorties de journal au format JSON

Les exemples suivants montrent comment les différentes sorties de journal générées à l'aide console des méthodes avec des arguments uniques et multiples sont capturées dans CloudWatch Logs lorsque vous définissez le format de journal de votre fonction sur JSON.

Le premier exemple utilise la méthode console.error pour générer une chaîne simple.

Example Code de journalisation Node.js

```
export const handler = async (event) => {
```

```
  console.error("This is a warning message");
  ...
}
```

Example Enregistrement de journaux JSON

```
{
    "timestamp":"2025-11-01T00:21:51.358Z",
    "level":"ERROR",
    "message":"This is a warning message",
    "requestId":"93f25699-2cbf-4976-8f94-336a0aa98c6f"
}
```

Vous pouvez également générer des messages de journal structurés plus complexes en utilisant un ou plusieurs arguments avec les méthodes `console`. Dans l'exemple suivant, vous pouvez utiliser `console.log` pour générer deux paires clé-valeur avec un seul argument. Notez que le `"message"` champ de l'objet JSON envoyé par Lambda à CloudWatch Logs n'est pas stringifié.

Example Code de journalisation Node.js

```
export const handler = async (event) => {
  console.log({data: 12.3, flag: false});
  ...
}
```

Example Enregistrement de journaux JSON

```
{
    "timestamp": "2025-12-08T23:21:04.664Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": {
        "data": 12.3,
        "flag": false
    }
}
```

Dans l'exemple suivant, vous utilisez à nouveau la méthode `console.log` pour créer une sortie de journal. Cette fois, la méthode prend deux arguments, une carte contenant deux paires clé-valeur et une chaîne d'identification. Notez que dans ce cas, parce que vous avez fourni deux arguments, Lambda stringifie le champ `"message"`.

Example Code de journalisation Node.js

```
export const handler = async (event) => {
  console.log('Some object - ', {data: 12.3, flag: false});
  ...
}
```

Example Enregistrement de journaux JSON

```
{
    "timestamp": "2025-12-08T23:21:04.664Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": "Some object -  { data: 12.3, flag: false }"
}
```

Lambda attribue les sorties générées à l'aide de `console.log` du niveau de journalisation INFO.

Le dernier exemple montre comment les objets d'erreur peuvent être générés dans CloudWatch Logs à l'aide `console` des méthodes. Notez que lorsque vous enregistrez des objets d'erreur à l'aide de plusieurs arguments, Lambda ajoute les champs `errorMessage`, `errorType` et `stackTrace` à la sortie du journal.

Example Code de journalisation Node.js

```
export const handler = async (event) => {
  let e1 = new ReferenceError("some reference error");
  let e2 = new SyntaxError("some syntax error");
  console.log(e1);
  console.log("errors logged - ", e1, e2);
};
```

Example Enregistrement de journaux JSON

```
{
    "timestamp": "2025-12-08T23:21:04.632Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": {
        "errorType": "ReferenceError",
        "errorMessage": "some reference error",
        "stackTrace": [
```

```
            "ReferenceError: some reference error",
            "    at Runtime.handler (file:///var/task/index.mjs:3:12)",
            "    at Runtime.handleOnceNonStreaming (file:///var/runtime/
index.mjs:1173:29)"
        ]
    }
}

{
    "timestamp": "2025-12-08T23:21:04.646Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": "errors logged -  ReferenceError: some reference error
\n    at Runtime.handler (file:///var/task/index.mjs:3:12)\n    at
 Runtime.handleOnceNonStreaming
    (file:///var/runtime/index.mjs:1173:29) SyntaxError: some syntax
 error\n    at Runtime.handler (file:///var/task/index.mjs:4:12)\n    at
 Runtime.handleOnceNonStreaming
    (file:///var/runtime/index.mjs:1173:29)",
    "errorType": "ReferenceError",
    "errorMessage": "some reference error",
    "stackTrace": [
        "ReferenceError: some reference error",
        "    at Runtime.handler (file:///var/task/index.mjs:3:12)",
        "    at Runtime.handleOnceNonStreaming (file:///var/runtime/index.mjs:1173:29)"
    ]
}
```

Lors de la journalisation de plusieurs types d'erreur, les champs supplémentaires `errorMessage`, `errorType`, et `stackTrace` sont extraits du premier type d'erreur fourni à la méthode `console`.

Utilisation de bibliothèques clientes au format métrique intégré (EMF) avec des journaux JSON structurés

AWS fournit des bibliothèques clientes open source pour Node.js que vous pouvez utiliser pour créer des journaux au format métrique intégré (EMF). Si vous avez des fonctions existantes qui utilisent ces bibliothèques et que vous modifiez le format de journal de votre fonction en JSON, il est CloudWatch possible que les métriques émises par votre code ne soient plus reconnues.

Si votre code émet actuellement des journaux EMF directement en utilisant `console.log` ou en utilisant Powertools for AWS Lambda (TypeScript), vous ne pourrez pas CloudWatch non plus les analyser si vous modifiez le format de journal de votre fonction en JSON.

> **Important**
>
> Pour vous assurer que les journaux EMF de vos fonctions continuent d'être correctement analysés CloudWatch, mettez à jour vos bibliothèques EMF et Powertools pour les AWS Lambda bibliothèques avec les dernières versions. Si vous passez au format de journal JSON, nous vous recommandons également d'effectuer des tests pour garantir la compatibilité avec les métriques intégrées de votre fonction. Si votre code émet des journaux EMF directement à l'aide de `console.log`, modifiez votre code pour qu'il affiche directement ces métriques vers `stdout`, comme indiqué dans l'exemple de code suivant.

Example code émettant des métriques intégrées pour **`stdout`**

```
process.stdout.write(JSON.stringify(
    {
        "_aws": {
            "Timestamp": Date.now(),
            "CloudWatchMetrics": [{
                "Namespace": "lambda-function-metrics",
                "Dimensions": [["functionVersion"]],
                "Metrics": [{
                    "Name": "time",
                    "Unit": "Milliseconds",
                    "StorageResolution": 60
                }]
            }]
        },
        "functionVersion": "$LATEST",
        "time": 100,
        "requestId": context.awsRequestId
    }
) + "\n")
```

Utilisation du filtrage au niveau du journal avec Node.js

AWS Lambda Pour filtrer les journaux de votre application en fonction de leur niveau de journalisation, votre fonction doit utiliser des journaux au format JSON. Vous pouvez effectuer cette opération de deux façons :

- Créez des sorties de journal à l'aide des méthodes de console standard et configurez votre fonction pour utiliser le format de journal JSON. AWS Lambda filtre ensuite les sorties de votre journal à l'aide de la paire clé-valeur « niveau » de l'objet JSON décrit dansthe section called “Utilisation de journaux JSON structurés avec Node.js”. Pour savoir comment configurer le format de journal de votre fonction, consultez the section called “Configuration de commandes de journalisation avancées pour votre fonction Lambda”.
- Utilisez une autre bibliothèque ou méthode de journalisation pour créer des journaux structurés JSON dans votre code qui incluent une paire clé-valeur « niveau » définissant le niveau de sortie du journal. Par exemple, vous pouvez utiliser Powertools pour générer des sorties AWS Lambda de journal structurées JSON à partir de votre code. Consultez the section called “Logging” pour en savoir plus sur l’utilisation de Powertools avec l’exécution Node.js.

  Pour que Lambda puisse filtrer les journaux de votre fonction, vous devez également inclure une paire `"timestamp"` clé-valeur dans la sortie de votre journal JSON. L’heure doit être spécifiée dans un format d’horodatage RFC 3339 valide. Si vous ne fournissez pas d’horodatage valide, Lambda attribuera au journal le niveau INFO et ajoutera un horodatage pour vous.

Lorsque vous configurez votre fonction pour utiliser le filtrage au niveau des journaux, vous sélectionnez le niveau de journaux que vous souhaitez envoyer AWS Lambda à CloudWatch Logs parmi les options suivantes :

Niveau de journalisation	Utilisation standard
TRACE (le plus détaillé)	Les informations les plus précises utilisées pour tracer le chemin d’exécution de votre code
DEBUG	Informations détaillées pour le débogage du système
INFO	Messages qui enregistrent le fonctionnement normal de votre fonction
WARN	Messages relatifs à des erreurs potentielles susceptibles d’entraîner un comportement inattendu si elles ne sont pas traitées

Niveau de journalisation	Utilisation standard
ERROR	Messages concernant les problèmes qui empêchent le code de fonctionner comme prévu
FATAL (moindre détail)	Messages relatifs à des erreurs graves entraînant l'arrêt du fonctionnement de l'application

Lambda envoie les journaux du niveau sélectionné et d'un niveau inférieur à. CloudWatch Par exemple, si vous configurez un niveau de journalisation WARN, Lambda envoie des journaux correspondant aux niveaux WARN, ERROR et FATAL.

Affichage des journaux dans la console Lambda

Vous pouvez utiliser la console Lambda pour afficher la sortie du journal après avoir invoqué une fonction Lambda.

Si votre code peut être testé à partir de l'éditeur Code intégré, vous trouverez les journaux dans les résultats d'exécution. Lorsque vous utilisez la fonctionnalité de test de console pour invoquer une fonction, vous trouverez Sortie du journal dans la section Détails.

Afficher les journaux dans la CloudWatch console

Vous pouvez utiliser la CloudWatch console Amazon pour consulter les journaux de toutes les invocations de fonctions Lambda.

Pour afficher les journaux sur la CloudWatch console

1. Ouvrez la page Groupes de journaux sur la CloudWatch console.
2. Choisissez le groupe de journaux pour votre fonction (***your-function-name***/aws/lambda/).
3. Choisissez un flux de journaux.

Chaque flux de journal correspond à une instance de votre fonction. Un flux de journal apparaît lorsque vous mettez à jour votre fonction Lambda et lorsque des instances supplémentaires sont créées pour gérer les appels simultanés. Pour trouver les journaux d'un appel spécifique, nous vous

recommandons d'instrumenter votre fonction avec. AWS X-Ray X-Ray enregistre des détails sur la demande et le flux de journaux dans le suivi.

Afficher les journaux à l'aide de AWS Command Line Interface (AWS CLI)

AWS CLI Il s'agit d'un outil open source qui vous permet d'interagir avec les AWS services à l'aide de commandes dans votre interface de ligne de commande. Pour effectuer les étapes de cette section, vous devez disposer de la version 2 de l'AWS CLI.

Vous pouvez utiliser AWS CLI pour récupérer les journaux d'une invocation à l'aide de l'option de commande `--log-type`. La réponse inclut un champ `LogResult` qui contient jusqu'à 4 Ko de journaux codés en base64 provenant de l'invocation.

Example récupérer un ID de journal

L'exemple suivant montre comment récupérer un ID de journal à partir du champ `LogResult` d'une fonction nommée `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Vous devriez voir la sortie suivante:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example décoder les journaux

Dans la même invite de commandes, utilisez l'utilitaire `base64` pour décoder les journaux. L'exemple suivant montre comment récupérer les journaux encodés en base64 pour `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-`

`in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

Vous devriez voir la sortie suivante :

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

L'utilitaire `base64` est disponible sous Linux, macOS et Ubuntu sous Windows. Les utilisateurs de macOS auront peut-être besoin d'utiliser `base64 -D`.

Example Script get-logs.sh

Dans la même invite de commandes, utilisez le script suivant pour télécharger les cinq derniers événements de journalisation. Le script utilise `sed` pour supprimer les guillemets du fichier de sortie et attend 15 secondes pour permettre la mise à disposition des journaux. La sortie comprend la réponse de Lambda, ainsi que la sortie de la commande `get-log-events`.

Copiez le contenu de l'exemple de code suivant et enregistrez-le dans votre répertoire de projet Lambda sous `get-logs.sh`.

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS et Linux (uniquement)

Dans la même invite de commandes, les utilisateurs macOS et Linux peuvent avoir besoin d'exécuter la commande suivante pour s'assurer que le script est exécutable.

```
chmod -R 755 get-logs.sh
```

Example récupérer les cinq derniers événements de journal

Dans la même invite de commande, exécutez le script suivant pour obtenir les cinq derniers événements de journalisation.

```
./get-logs.sh
```

Vous devriez voir la sortie suivante:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
```

```
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


Suppression de journaux

Les groupes de journaux ne sont pas supprimés automatiquement quand vous supprimez une fonction. Pour éviter de stocker des journaux indéfiniment, supprimez le groupe de journaux ou configurez une période de conservation à l'issue de laquelle les journaux sont supprimés automatiquement.

Instrumentation du code Node.js dans AWS Lambda

Lambda s'intègre pour vous aider AWS X-Ray à suivre, à déboguer et à optimiser les applications Lambda. Vous pouvez utiliser X-Ray pour suivre une demande lorsque celle-ci parcourt les ressources de votre application, qui peuvent inclure des fonctions Lambda et d'autres services AWS .

Pour envoyer des données de suivi à X-Ray, vous pouvez utiliser l'une des deux bibliothèques SDK suivantes :

- AWS Distro for OpenTelemetry (ADOT) — Une distribution sécurisée, prête pour la production et AWS prise en charge du SDK (). OpenTelemetry OTel
- Kit SDK AWS X-Ray pour Node.js – Un kit SDK permettant de générer et d'envoyer des données de suivi à X-Ray.

Chacune d'entre elles SDKs propose des moyens d'envoyer vos données de télémétrie au service X-Ray. Vous pouvez ensuite utiliser X-Ray pour afficher, filtrer et avoir un aperçu des métriques de performance de votre application, afin d'identifier les problèmes et les occasions d'optimiser votre application.

> **Important**
>
> Les outils X-Ray et Powertools pour AWS Lambda SDKs font partie d'une solution d'instrumentation étroitement intégrée proposée par AWS. Les couches ADOT Lambda font partie d'une norme industrielle pour l'instrumentation de traçage qui collecte plus de données en général, mais qui peut ne pas convenir à tous les cas d'utilisation. Vous pouvez implémenter le end-to-end traçage dans X-Ray en utilisant l'une ou l'autre solution. Pour en savoir plus sur le choix entre les deux, consultez Choosing between the AWS Distro for Open Telemetry and X-Ray. SDKs

Sections

- Utilisation d'ADOT pour instrumenter vos fonctions Node.js
- Utilisation du kit SDK X-Ray pour instrumenter vos fonctions Node.js
- Activation du suivi avec la console Lambda
- Activation du suivi avec l'API Lambda
- Activation du traçage avec CloudFormation

- Interprétation d’un suivi X-Ray
- Stockage des dépendances d’exécution dans une couche (kit SDK X-Ray)

Utilisation d'ADOT pour instrumenter vos fonctions Node.js

ADOT fournit des couches Lambda entièrement gérées qui regroupent tout ce dont vous avez besoin pour collecter des données de télémétrie à l'aide du SDK. OTel En consommant cette couche, vous pouvez instrumenter vos fonctions Lambda sans avoir à modifier le code de fonction. Vous pouvez également configurer votre couche pour effectuer une initialisation personnalisée de OTel. Pour de plus amples informations, veuillez consulter Configuration personnalisée pour ADOT Collector sur Lambda dans la documentation ADOT.

Pour les exécutions python, vous pouvez ajouter leAWS couche Lambda gérée pour ADOT Javascriptpour instrumenter automatiquement vos fonctions. Pour obtenir des instructions détaillées sur la façon d'ajouter cette couche, consultez AWS Distro for OpenTelemetry Lambda JavaScript Support dans la documentation ADOT.

Utilisation du kit SDK X-Ray pour instrumenter vos fonctions Node.js

Pour enregistrer des détails sur les appels effectués par votre fonction Lambda à d’autres ressources de votre application, vous pouvez également utiliser le Kit SDK AWS X-Ray pour Node.js. Pour obtenir ce kit SDK, ajoutez le package `aws-xray-sdk-core` aux dépendances de votre application.

Example blank-nodejs/package.json

```
{
  "name": "blank-nodejs",
  "version": "1.0.0",
  "private": true,
  "devDependencies": {
    "jest": "29.7.0"
  },
  "dependencies": {
    "@aws-sdk/client-lambda": "3.345.0",
    "aws-xray-sdk-core": "3.5.3"
  },
  "scripts": {
    "test": "jest"
  }
```

```
}
```

Pour instrumenter les clients du AWS SDK dans la AWS SDK pour JavaScript v3, encapsulez l'instance du client avec la `captureAWSv3Client` méthode.

Example nodejs/function/indexblank-.js — Suivi d'un client AWS SDK

```
const AWSXRay = require('aws-xray-sdk-core');
const { LambdaClient, GetAccountSettingsCommand } = require('@aws-sdk/client-lambda');

// Create client outside of handler to reuse
const lambda = AWSXRay.captureAWSv3Client(new LambdaClient());

// Handler
exports.handler = async function(event, context) {
    event.Records.forEach(record => {
  ...
```

Le moteur d'exécution Lambda définit certaines variables d'environnement pour configurer le kit SDK X-Ray. Lambda définit par exemple `AWS_XRAY_CONTEXT_MISSING` à `LOG_ERROR` pour éviter de générer des erreurs d'environnement d'exécution à partir du kit SDK X-Ray. Pour définir une stratégie manquante de contexte personnalisée, remplacez la variable d'environnement dans votre configuration de fonction pour n'avoir aucune valeur, puis vous pouvez définir la stratégie manquante de contexte par programme.

Example Exemple de code d'initialisation

```
const AWSXRay = require('aws-xray-sdk-core');

// Configure the context missing strategy to do nothing
AWSXRay.setContextMissingStrategy(() => {});
```

Pour de plus amples informations, veuillez consulter the section called “Variables d’environnement”.

Une fois que vous avez ajouté les bonnes dépendances et effectué les modifications de code nécessaires, activez le suivi dans la configuration de votre fonction via la console Lambda ou l’API.

Activation du suivi avec la console Lambda

Pour activer/désactiver le traçage actif sur votre fonction Lambda avec la console, procédez comme suit :

Pour activer le traçage actif

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Choisissez Configuration, puis choisissez Outils de surveillance et d'opérations.
4. Dans Outils de surveillance supplémentaires, choisissez Modifier.
5. Sous Signaux CloudWatch d'application et AWS X-Ray sélectionnez Activer les traces de service Lambda.
6. Choisissez Enregistrer.

Activation du suivi avec l'API Lambda

Configurez le suivi sur votre fonction Lambda avec le AWS SDK AWS CLI or, utilisez les opérations d'API suivantes :

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

L'exemple de AWS CLI commande suivant active le suivi actif sur une fonction nommée my-function.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Le mode de suivi fait partie de la configuration spécifique de la version lorsque vous publiez une version de votre fonction. Vous ne pouvez pas modifier le mode de suivi sur une version publiée.

Activation du traçage avec CloudFormation

Pour activer le suivi d'une `AWS::Lambda::Function` ressource dans un CloudFormation modèle, utilisez la `TracingConfig` propriété.

Example function-inline.yml – Configuration du suivi

```
Resources:
```

```
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

Pour une `AWS::Serverless::Function` ressource AWS Serverless Application Model (AWS SAM), utilisez la `Tracing` propriété.

Example template.yml – Configuration du suivi

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

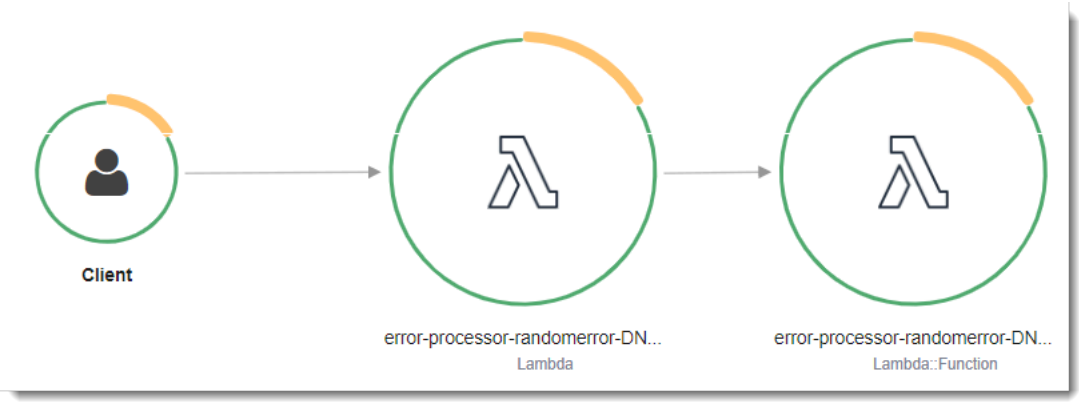
Interprétation d'un suivi X-Ray

Votre fonction a besoin d'une autorisation pour charger des données de suivi vers X-Ray. Lorsque vous activez le suivi actif dans la console Lambda, Lambda ajoute les autorisations requises au rôle d'exécution de votre fonction. Sinon, ajoutez la AWSXRayDaemonWriteAccesspolitique au rôle d'exécution.

Une fois que vous avez configuré le suivi actif, vous pouvez observer des demandes spécifiques via votre application. Le graphique de services X-Ray affiche des informations sur votre application et tous ses composants. L'exemple suivant montre une application dotée de deux fonctions. La fonction principale traite les événements et renvoie parfois des erreurs. La deuxième fonction située en haut traite les erreurs qui apparaissent dans le groupe de journaux de la première et utilise le AWS SDK pour appeler X-Ray, Amazon Simple Storage Service (Amazon S3) et Amazon Logs. CloudWatch

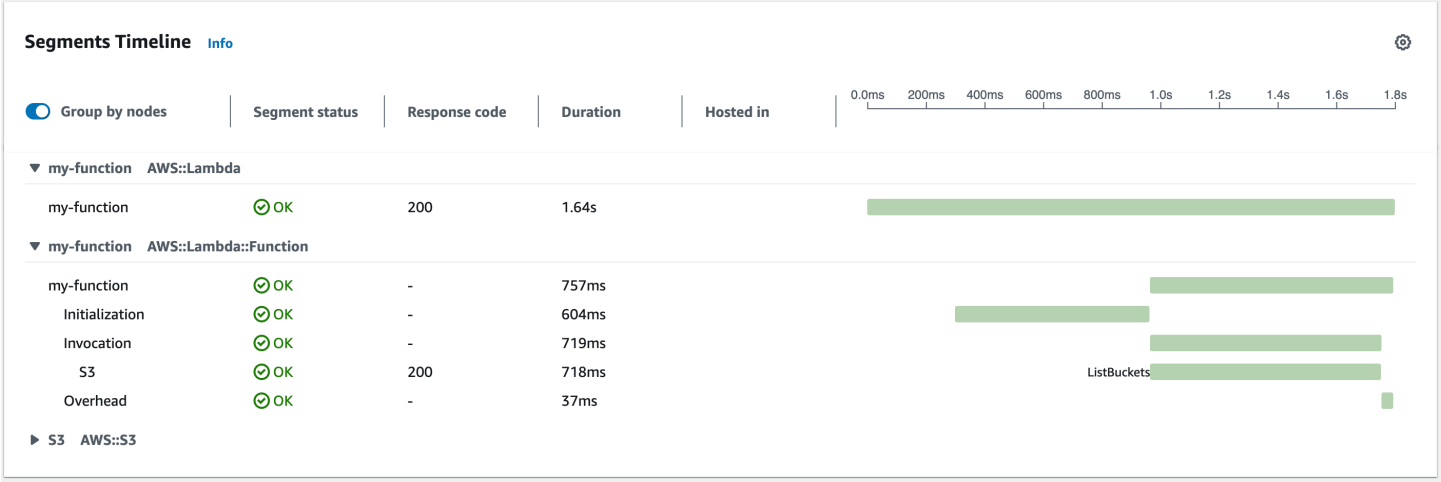
X-Ray ne trace pas toutes les requêtes vers votre application. X-Ray applique un algorithme d'échantillonnage pour s'assurer que le suivi est efficace, tout en fournissant un échantillon représentatif de toutes les demandes. Le taux d'échantillonnage est 1 demande par seconde et 5 % de demandes supplémentaires. Vous ne pouvez pas configurer ce taux d'échantillonnage X-Ray pour vos fonctions.

Dans X-Ray, un suivi enregistre des informations sur une demande traitée par un ou plusieurs services. Lambda enregistre deux segments par suivi, ce qui a pour effet de créer deux nœuds sur le graphique du service. L'image suivante met en évidence ces deux nœuds :



Le premier nœud sur la gauche représente le service Lambda qui reçoit la demande d'invocation. Le deuxième nœud représente votre fonction Lambda spécifique. L'exemple suivant illustre une trace avec ces deux segments. Les deux sont nommés my-function, mais l'un a pour origine `AWS::Lambda` et l'autre a pour origine `AWS::Lambda::Function`. Si le segment `AWS::Lambda` affiche une erreur, cela signifie que le service Lambda a rencontré un problème. Si le segment

`AWS::Lambda::Function` affiche une erreur, cela signifie que votre fonction a rencontré un problème.



Cet exemple développe le segment `AWS::Lambda::Function` pour afficher ses trois sous-segments.

> **Note**
>
> AWS met actuellement en œuvre des modifications du service Lambda. En raison de ces modifications, vous pouvez constater des différences mineures entre la structure et le contenu des messages du journal système et des segments de suivi émis par les différentes fonctions Lambda de votre Compte AWS.
> L'exemple de suivi présenté ici illustre le segment de fonction à l'ancienne. Les différences entre les segments à l'ancienne et de style moderne sont décrites dans les paragraphes suivants.
> Ces modifications seront mises en œuvre au cours des prochaines semaines, et toutes les fonctions, Régions AWS sauf en Chine et dans les GovCloud régions, seront transférées pour utiliser le nouveau format des messages de journal et des segments de trace.

Le segment de fonction à l'ancienne contient les sous-segments suivants :

- Initialization (Initialisation) : représente le temps passé à charger votre fonction et à exécuter le code d'initialisation. Ce sous-segment apparaît pour le premier événement traité par chaque instance de votre fonction.
- Invocation – Représente le temps passé à exécuter votre code de gestionnaire.

- Overhead (Travail supplémentaire) – Représente le temps que le fichier d'exécution Lambda passe à se préparer à gérer l'événement suivant.

Le segment de fonction de style moderne ne contient pas de sous-segment `Invocation`. À la place, les sous-segments du client sont directement rattachés au segment de fonction. Pour plus d'informations sur la structure des segments de fonction à l'ancienne et de style moderne, consultez the section called "Comprendre les suivis X-Ray".

Vous pouvez également utiliser des clients HTTP, enregistrer des requêtes SQL et créer des sous-segments personnalisés avec des annotations et des métadonnées. Pour plus d'informations, consultez Kit SDK AWS X-Ray pour Node.js dans le AWS X-Ray Guide du développeur.

> **Tarification**
>
> Vous pouvez utiliser le X-Ray Tracing gratuitement chaque mois jusqu'à une certaine limite dans le cadre du niveau AWS gratuit. Au-delà de ce seuil, X-Ray facture le stockage et la récupération du suivi. Pour en savoir plus, consultez Pricing AWS X-Ray (Tarification).

Stockage des dépendances d'exécution dans une couche (kit SDK X-Ray)

Si vous utilisez le SDK X-Ray pour instrumenter le code de fonction des clients du AWS SDK, votre package de déploiement peut devenir très volumineux. Pour éviter de charger des dépendances d'environnement d'exécution chaque fois que vous mettez à jour votre code de fonction, empaquetez le kit SDK X-Ray dans une couche Lambda.

L'exemple suivant montre une ressource `AWS::Serverless::LayerVersion` qui stocke le Kit SDK AWS X-Ray pour Node.js.

Example template.yml : couche de dépendances

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: function/.
      Tracing: Active
      Layers:
        - !Ref libs
```

```
    ...
libs:
  Type: AWS::Serverless::LayerVersion
  Properties:
    LayerName: blank-nodejs-lib
    Description: Dependencies for the blank sample app.
    ContentUri: lib/.
    CompatibleRuntimes:
      - nodejs24.x
```

Avec cette configuration, vous ne mettez à jour les fichiers de couche de bibliothèque que si vous modifiez vos dépendances d'exécution. Étant donné que le package de déploiement de la fonction contient uniquement votre code, cela peut contribuer à réduire les temps de chargement.

La création d'une couche de dépendances nécessite des modifications de génération pour créer l'archive des couches avant le déploiement. Pour un exemple fonctionnel, consultez l'exemple d'application blank-nodejs.

Création de fonctions Lambda avec TypeScript

Vous pouvez utiliser le moteur d'exécution Node.js pour exécuter TypeScript du code dans AWS Lambda. Étant donné que Node.js n'exécute pas TypeScript le code de manière native, vous devez d'abord le transpiler dans TypeScript . JavaScript Utilisez ensuite les JavaScript fichiers pour déployer votre code de fonction sur Lambda. Votre code s'exécute dans un environnement qui inclut le AWS SDK pour JavaScript, avec les informations d'identification d'un rôle Gestion des identités et des accès AWS (IAM) que vous gérez. Pour en savoir plus sur les versions du kit SDK incluses dans les environnements d'exécution Node.js, consultez the section called “Versions du SDK incluses dans l'environnement d'exécution”.

Lambda prend en charge les environnements d'exécution Node.js suivants.

Name	Identifiant	Système d'exploitation	Date d'obsolescence	Créer la fonction de blocage	Mettre à jour la fonction de blocage
Node.js 24	`nodejs24.x`	Amazon Linux 2	30 avril 2028	1 juin 2028	1 juillet 2028
Node.js 22	`nodejs22.x`	Amazon Linux 2	30 avril 2027	1 juin 2027	1 juillet 2027
Node.js 20	`nodejs20.x`	Amazon Linux 2	30 avril 2026	31 août 2026	30 sept. 2026

Rubriques

- Configuration d'un environnement TypeScript de développement
- Définitions de type pour Lambda
- Définissez le gestionnaire de fonctions Lambda dans TypeScript
- Déployez TypeScript du code transpilé dans Lambda avec des archives de fichiers .zip
- Déployez le code TypeScript transpilé dans Lambda avec des images du conteneur
- Utilisation de l'objet de contexte Lambda pour récupérer les informations relatives aux fonctions TypeScript
- Enregistrez et surveillez les fonctions TypeScript Lambda

- TypeScript Code de suivi dans AWS Lambda

Configuration d'un environnement TypeScript de développement

Utilisez un environnement de développement intégré (IDE) local ou un éditeur de texte pour écrire votre code de TypeScript fonction. Vous ne pouvez pas créer de TypeScript code sur la console Lambda.

Vous pouvez utiliser esbuild ou le TypeScript compilateur (`tsc`) de Microsoft pour transpiler votre TypeScript code dans. JavaScript AWS Serverless Application Model (AWS SAM) et AWS Cloud Development Kit (AWS CDK) utilisent tous deux esbuild.

Tenez compte des éléments suivants lorsque vous utilisez esbuild :

- Il y a plusieurs TypeScript mises en garde.
- Vous devez configurer vos paramètres de TypeScript transpilation pour qu'ils correspondent à l'environnement d'exécution Node.js que vous prévoyez d'utiliser. Pour plus d'informations, consultez Cible dans la documentation d'esbuild. Pour un exemple de fichier tsconfig.json qui montre comment cibler une version spécifique de Node.js prise en charge par Lambda, reportez-vous au référentiel. TypeScript GitHub
- esbuild n'effectue pas de vérifications du type. Pour vérifier les types, utilisez le compilateur `tsc`. Exécutez `tsc -noEmit` ou ajoutez un paramètre `"noEmit"` dans votre fichier tsconfig.json comme illustré dans l'exemple suivant. Cela permet de ne pas `tsc` émettre de JavaScript fichiers. Après avoir vérifié les types, utilisez esbuild pour convertir les TypeScript fichiers en JavaScript.

Example tsconfig.json

```
{
 "compilerOptions": {
   "target": "es2020",
   "strict": true,
   "preserveConstEnums": true,
   "noEmit": true,
   "sourceMap": false,
   "module":"commonjs",
   "moduleResolution":"node",
   "esModuleInterop": true,
   "skipLibCheck": true,
```

```
    "forceConsistentCasingInFileNames": true,
    "isolatedModules": true,
  },
  "exclude": ["node_modules", "**/*.test.ts"]
}
```

Définitions de type pour Lambda

Le package @types/aws-lambda contient les définitions de type pour les fonctions Lambda. Installez ce package lorsque votre fonction utilise l'un des éléments suivants :

- Sources AWS d'événements courantes, telles que :
  - `APIGatewayProxyEvent` : pour les intégrations de proxy Amazon API Gateway
  - `SNSEvent` : pour les notifications Amazon Simple Notification Service
  - `SQSEvent` : pour les messages Amazon Simple Queue Service
  - `S3Event` : pour les événements de déclencheur S3
  - `DynamoDBStreamEvent` : pour Amazon DynamoDB Streams
- L'objet Contexte Lambda
- Le modèle du gestionnaire de rappels

Pour ajouter les définitions de type Lambda à votre fonction, installez `@types/aws-lambda` en tant que dépendance de développement :

```
npm install -D @types/aws-lambda
```

Importez ensuite les types depuis `aws-lambda` :

```
import { Context, S3Event, APIGatewayProxyEvent } from 'aws-lambda';

export const handler = async (event: S3Event, context: Context) => {
    // Function code
};
```

L'instruction `import ... from 'aws-lambda'` importe les définitions de type. Elle n'importe pas le package npm `aws-lambda`, qui est un outil tiers indépendant. Pour plus d'informations, consultez aws-lambda dans le référentiel. DefinitelyTyped GitHub

> **Note**
>
> Vous n'avez pas besoin de @types/aws-lambda lorsque vous utilisez vos propres définitions de type personnalisées. Pour un exemple de fonction qui définit son propre type pour un objet d'événement, consultez Exemple de code de TypeScript fonction Lambda.

Définissez le gestionnaire de fonctions Lambda dans TypeScript

Le gestionnaire de fonction Lambda est la méthode dans votre code de fonction qui traite les événements. Lorsque votre fonction est invoquée, Lambda exécute la méthode du gestionnaire. Votre fonction s'exécute jusqu'à ce que le gestionnaire renvoie une réponse, se ferme ou expire.

Cette page explique comment utiliser les gestionnaires de fonctions Lambda dans TypeScript, notamment les options de configuration du projet, les conventions de dénomination et les meilleures pratiques. Cette page inclut également un exemple de fonction TypeScript Lambda qui collecte des informations relatives à une commande, produit un reçu sous forme de fichier texte et place ce fichier dans un bucket Amazon Simple Storage Service (Amazon S3). Pour plus d'informations sur le déploiement de votre fonction après l'avoir écrite, consultez the section called "Déployez des archives de fichiers .zip" ou the section called "Déployez les images du conteneur".

Rubriques

- Configuration de votre TypeScript projet
- Exemple de code de TypeScript fonction Lambda
- Modules CommonJS et ES
- Initialisation de Node.js
- Convention de nommage du gestionnaire
- Définition et accès à l'objet d'événement d'entrée
- Modèles de gestionnaire valides pour les fonctions TypeScript
- Utilisation du SDK pour la JavaScript version 3 dans votre gestionnaire
- Accès aux variables d'environnement
- Utilisation de l'état global
- Meilleures pratiques en matière de code pour les fonctions TypeScript Lambda

Configuration de votre TypeScript projet

Utilisez un environnement de développement intégré (IDE) local ou un éditeur de texte pour écrire votre code de TypeScript fonction. Vous ne pouvez pas créer de TypeScript code sur la console Lambda.

Il existe plusieurs manières d'initialiser un projet TypeScript Lambda. Par exemple, vous pouvez créer un projet en utilisant npm, créer une application AWS SAM ou créer une application AWS CDK. Pour créer un projet avec npm :

```
npm init
```

Le code de votre fonction se trouve dans un `.ts` fichier que vous transpirez dans un JavaScript fichier au moment de la création. Vous pouvez utiliser esbuild ou le TypeScript compilateur (`tsc`) de Microsoft pour transpiler votre TypeScript code dans. JavaScript Pour utiliser esbuild, ajoutez-le en tant que dépendance de développement :

```
npm install -D esbuild
```

Un projet de fonction TypeScript Lambda typique suit cette structure générale :

```
/project-root
  ### index.ts - Contains main handler
  ### dist/ - Contains compiled JavaScript
  ### package.json - Project metadata and dependencies
  ### package-lock.json - Dependency lock file
  ### tsconfig.json - TypeScript configuration
  ### node_modules/ - Installed dependencies
```

Exemple de code de TypeScript fonction Lambda

L'exemple de code de fonction Lambda suivant prend en compte les informations relatives à une commande, produit un reçu sous forme de fichier texte et place ce fichier dans un compartiment Amazon S3. Cet exemple définit un type d'événement personnalisé (`OrderEvent`). Pour savoir comment importer des définitions de type pour les sources d' AWS événements, consultez la section Définitions de type pour Lambda.

> **Note**
>
> Cet exemple utilise un gestionnaire de modules ES. Lambda prend en charge à la fois le module ES et les gestionnaires CommonJS. Pour de plus amples informations, veuillez consulter Modules CommonJS et ES.

Example Fonction Lambda index.ts

```
import { S3Client, PutObjectCommand } from '@aws-sdk/client-s3';

// Initialize the S3 client outside the handler for reuse
const s3Client = new S3Client();

// Define the shape of the input event
type OrderEvent = {
    order_id: string;
    amount: number;
    item: string;
}

/**
 * Lambda handler for processing orders and storing receipts in S3.
 */
export const handler = async (event: OrderEvent): Promise<string> => {
    try {
        // Access environment variables
        const bucketName = process.env.RECEIPT_BUCKET;
        if (!bucketName) {
            throw new Error('RECEIPT_BUCKET environment variable is not set');
        }

        // Create the receipt content and key destination
        const receiptContent = `OrderID: ${event.order_id}\nAmount: $
${event.amount.toFixed(2)}\nItem: ${event.item}`;
        const key = `receipts/${event.order_id}.txt`;

        // Upload the receipt to S3
        await uploadReceiptToS3(bucketName, key, receiptContent);

        console.log(`Successfully processed order ${event.order_id} and stored receipt
 in S3 bucket ${bucketName}`);
        return 'Success';
    } catch (error) {
        console.error(`Failed to process order: ${error instanceof Error ?
 error.message : 'Unknown error'}`);
        throw error;
    }
};

/**
```

```
 * Helper function to upload receipt to S3
 */
async function uploadReceiptToS3(bucketName: string, key: string, receiptContent:
 string): Promise<void> {
    try {
        const command = new PutObjectCommand({
            Bucket: bucketName,
            Key: key,
            Body: receiptContent
        });

        await s3Client.send(command);
    } catch (error) {
        throw new Error(`Failed to upload receipt to S3: ${error instanceof Error ?
 error.message : 'Unknown error'}`);
    }
}
```

Ce fichier `index.ts` comprend les sections suivantes :

- Bloc `import` : utilisez ce bloc pour inclure les bibliothèques que votre fonction Lambda requiert, comme les clients du SDK AWS.
- Déclaration `const s3Client` : initialise un client Amazon S3 en dehors de la fonction du gestionnaire. Lambda exécute donc ce code pendant la phase d'initialisation, et le client est conservé pour être réutilisé lors de plusieurs appels.
- `type OrderEvent` : définit la structure de l'événement d'entrée attendu.
- `export const handler` : le gestionnaire de la fonction principal que Lambda invoque. Lorsque vous déployez votre fonction, spécifiez `index.handler` pour la propriété Gestionnaire. La valeur de la propriété `Handler` est le nom du fichier et le nom de la méthode du gestionnaire exporté, séparés par un point.
- Fonction `uploadReceiptToS3` : une fonction d'assistance référencée par la fonction du gestionnaire principal.

Pour que cette fonction fonctionne correctement, son rôle d'exécution doit autoriser l'action `s3:PutObject`. Assurez-vous également de définir la variable d'environnement `RECEIPT_BUCKET`. Après une invocation réussie, le compartiment Amazon S3 doit contenir un fichier de reçu.

Modules CommonJS et ES

Node.js prend en charge deux systèmes de modules : CommonJS et ECMAScript modules (modules ES). Lambda recommande d'utiliser les modules ES car ils prennent en charge l'attente de haut niveau, ce qui permet d'effectuer des tâches asynchrones lors de l'initialisation de l'environnement d'exécution.

Node.js traite les fichiers avec une extension de nom de `.cjs` fichier comme des modules CommonJS tandis qu'une `.mjs` extension désigne les modules ES. Par défaut, Node.js traite les fichiers portant l'extension de nom de `.js` fichier comme des modules CommonJS. Vous pouvez configurer Node.js pour traiter les `.js` fichiers comme des modules ES en spécifiant le `type` as `module` dans le `package.json` fichier de la fonction. Vous pouvez configurer Node.js dans Lambda pour détecter automatiquement si un `.js` fichier doit être traité comme CommonJS ou comme un module ES en ajoutant l'`–experimental-detect-module`indicateur à la `NODE_OPTIONS` variable d'environnement. Pour plus d'informations, consultez la section Fonctionnalités expérimentales de Node.js.

Les exemples suivants montrent des gestionnaires de fonctions écrits à l'aide de modules ES et de modules CommonJS. Les autres exemples de cette page utilisent tous des modules ES.

Initialisation de Node.js

Node.js utilise un I/O modèle non bloquant qui prend en charge des opérations asynchrones efficaces à l'aide d'une boucle d'événements. Par exemple, si Node.js effectue un appel réseau, la fonction continue de traiter les autres opérations sans bloquer une réponse réseau. Lorsque la réponse du réseau est reçue, elle est placée dans la file de rappel. Les tâches de la file d'attente sont traitées lorsque la tâche en cours est terminée.

Lambda recommande d'utiliser l'attente de haut niveau afin que les tâches asynchrones initiées lors de l'initialisation de l'environnement d'exécution soient terminées pendant l'initialisation. Les tâches asynchrones qui ne sont pas terminées lors de l'initialisation s'exécutent généralement lors du premier appel de fonction. Cela peut entraîner un comportement inattendu ou des erreurs. Par exemple, l'initialisation de votre fonction peut effectuer un appel réseau pour récupérer un paramètre depuis le Parameter Store AWS . Si cette tâche n'est pas terminée lors de l'initialisation, la valeur peut être nulle lors d'un appel. Il peut également y avoir un délai entre l'initialisation et l'appel, ce qui peut déclencher des erreurs lors d'opérations sensibles au facteur temps. En particulier, les appels de AWS service peuvent s'appuyer sur des signatures de demande urgentes, ce qui entraîne l'échec des appels de service si l'appel n'est pas terminé pendant la phase d'initialisation. L'exécution

des tâches lors de l'initialisation améliore généralement les performances de démarrage à froid et les performances d'appel d'abord lors de l'utilisation de la concurrence provisionnée. Pour plus d'informations, consultez notre article de blog Using Node.js ES modules and top-level await in AWS Lambda.

Convention de nommage du gestionnaire

Lorsque vous configurez une fonction, la valeur du paramètre Gestionnaire est le nom du fichier et le nom de la méthode du gestionnaire exporté, séparés par un point. La valeur par défaut des fonctions créées dans la console et dans les exemples de ce guide est `index.handler`. Cela indique la méthode `handler` qui est exportée à partir du fichier `index.js` ou `index.mjs`.

Si vous créez une fonction dans la console en utilisant un nom de fichier ou un nom de gestionnaire de fonction différent, vous devez modifier le nom du gestionnaire par défaut.

Pour modifier le nom du gestionnaire de fonction (console)

1. Ouvrez la page Fonctions de la console Lambda et choisissez votre fonction.
2. Cliquez sur l'onglet Code.
3. Faites défiler l'écran jusqu'au volet Paramètres d'exécution et choisissez Modifier.
4. Dans Gestionnaire, saisissez le nouveau nom de votre gestionnaire de fonction.
5. Choisissez Enregistrer.

Définition et accès à l'objet d'événement d'entrée

JSON est le format d'entrée le plus courant et standard pour les fonctions Lambda. Dans cet exemple, la fonction exige une entrée similaire à l'exemple suivant :

```
{
    "order_id": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

Lorsque vous utilisez des fonctions Lambda dans TypeScript, vous pouvez définir la forme de l'événement d'entrée à l'aide d'un type ou d'une interface. Dans cet exemple, nous définissons la structure de l'événement à l'aide d'un type :

```
type OrderEvent = {
    order_id: string;
    amount: number;
    item: string;
}
```

Après avoir défini le type ou l'interface, utilisez-le dans la signature de votre gestionnaire pour garantir la sécurité du type :

```
export const handler = async (event: OrderEvent): Promise<string> => {
```

Lors de la compilation, TypeScript vérifie que l'objet d'événement contient les champs obligatoires avec les types corrects. Par exemple, le TypeScript compilateur signale une erreur si vous essayez de l'utiliser `event.order_id` sous forme de nombre ou `event.amount` de chaîne.

Modèles de gestionnaire valides pour les fonctions TypeScript

Nous vous recommandons d'utiliser async/await pour déclarer le gestionnaire de fonctions au lieu d'utiliser des rappels. Async/await est un moyen concis et lisible d'écrire du code asynchrone, sans avoir besoin de rappels imbriqués ou de promesses enchaînées. Avec async/await, vous pouvez écrire du code qui se lit comme du code synchrone, tout en étant asynchrone et non bloquant.

Les exemples de cette section utilisent le type `S3Event`. Cependant, vous pouvez utiliser n'importe quel autre type d' AWS événement dans le package @types /aws-lambda ou définir votre propre type d'événement. Pour utiliser des types provenant de @types/aws-lambda :

1. Ajoutez le package @types/aws-lambda en tant que dépendance de développement :

   ```
   npm install -D @types/aws-lambda
   ```

2. Importez les types dont vous avez besoin, comme `Context`, `S3Event` ou `Callback`.

gestionnaires de fonctions asynchrones (recommandé)

Le mot-clé `async` marque une fonction comme étant asynchrone, et le mot-clé `await` met en pause l'exécution de la fonction jusqu'à ce qu'une `Promise` soit résolue. Le gestionnaire accepte les arguments suivants :

- `event` : contient les données d'entrée transmises à votre fonction.

- `context` : contient des informations sur l'invocation, la fonction et l'environnement d'exécution. Pour de plus amples informations, veuillez consulter Utilisation de l'objet de contexte Lambda pour récupérer les informations relatives aux fonctions TypeScript .

Voici les signatures valides pour le async/await modèle :

```
export const handler = async (event: S3Event): Promise<void> => { };
```

```
export const handler = async (event: S3Event, context: Context): Promise<void> => { };
```

> **Note**
>
> Lorsque vous traitez des tableaux d'éléments de manière asynchrone, assurez-vous d'utiliser await avec `Promise.all` pour vous assurer que toutes les opérations sont terminées. Des méthodes comme `forEach` n'attendent pas la fin des rappels asynchrones. Pour plus d'informations, consultez Array.prototype.forEach() dans la documentation Mozilla.

Gestionnaires de fonctions synchrones

Lorsque votre fonction n'exécute aucune tâche asynchrone, vous pouvez utiliser un gestionnaire de fonctions synchrones en utilisant l'une des signatures de fonction suivantes :

```
export const handler = (event: S3Event): void => { };
```

```
export const handler = (event: S3Event, context: Context): void => { };
```

Gestionnaires de fonctions de diffusion des réponses

Lambda prend en charge le streaming des réponses avec Node.js. Les gestionnaires de fonctions de streaming de réponses utilisent le décorateur AWSLambda.streamifyResponse () et prennent 3 paramètres : event, responseStream et context. La signature de la fonction est la suivante :

```
export const handler = awslambda.streamifyResponse(async (event: APIGatewayProxyEvent,
 responseStream: NodeJS.WritableStream, context: Context) => { });
```

Pour plus d'informations, consultez Streaming de réponses pour les fonctions Lambda.

Gestionnaires de fonctions basés sur le rappel

> **Note**
>
> Les gestionnaires de fonctions basés sur le rappel ne sont pris en charge que jusqu'à Node.js 22. À partir de Node.js 24, les tâches asynchrones doivent être implémentées à l'aide de gestionnaires de fonctions asynchrones.

Les gestionnaires de fonctions basés sur le rappel peuvent utiliser les arguments d'événement, de contexte et de rappel. L'argument de rappel attend une `Error` et une réponse, qui doit être sérialisable en JSON.

Voici la signature valide pour le modèle de gestionnaire de rappel :

```
export const handler = (event: S3Event, context: Context, callback: Callback<void>):
 void => { };
```

La fonction continue à s'exécuter jusqu'à ce que la boucle d'événement soit vide ou que la fonction s'arrête. La réponse n'est pas envoyée à l'appelant tant que toutes les tâches d'événement de boucle ne sont pas terminées. Si la fonction expire, une erreur est renvoyée à la place. Vous pouvez configurer le moteur d'exécution pour envoyer la réponse immédiatement en définissant le contexte. callbackWaitsForEmptyEventLoopà faux.

Example TypeScript fonction avec rappel

L'exemple suivant utilise `APIGatewayProxyCallback`, un type de rappel spécialisé spécifique aux intégrations API Gateway. La plupart des sources d' AWS événements utilisent le `Callback` type générique indiqué dans les signatures ci-dessus.

```
import { Context, APIGatewayProxyCallback, APIGatewayEvent } from 'aws-lambda';

export const lambdaHandler = (event: APIGatewayEvent, context: Context, callback:
 APIGatewayProxyCallback): void => {
    console.log(`Event: ${JSON.stringify(event, null, 2)}`);
    console.log(`Context: ${JSON.stringify(context, null, 2)}`);
    callback(null, {
        statusCode: 200,
        body: JSON.stringify({
            message: 'hello world',
```

```
        }),
    });
};
```

Utilisation du SDK pour la JavaScript version 3 dans votre gestionnaire

Vous utiliserez souvent les fonctions Lambda pour interagir avec d'autres AWS ressources ou pour les mettre à jour. Le moyen le plus simple d'interagir avec ces ressources est d'utiliser AWS SDK pour JavaScript. Tous les environnements d'exécution Lambda Node.js pris en charge incluent le SDK pour la version 3. JavaScript Toutefois, nous vous recommandons vivement d'inclure les clients AWS SDK dont vous avez besoin dans votre package de déploiement. Cela permet d'optimiser la rétrocompatibilité lors des futures mises à jour de l'environnement d'exécution Lambda.

Pour ajouter des dépendances au SDK à votre fonction, utilisez la commande `npm install` correspondant aux clients SDK spécifiques dont vous avez besoin. Dans l'exemple de code, nous avons utilisé le client Amazon S3. Ajoutez cette dépendance en exécutant la commande suivante dans le répertoire qui contient votre fichier `package.json` :

```
npm install @aws-sdk/client-s3
```

Dans le code de fonction, importez le client et les commandes dont vous avez besoin, comme le montre l'exemple de fonction :

```
import { S3Client, PutObjectCommand } from '@aws-sdk/client-s3';
```

Initialisez ensuite un client Amazon S3 :

```
const s3Client = new S3Client();
```

Dans cet exemple, nous avons initialisé notre client Amazon S3 en dehors de la fonction de gestionnaire principale pour éviter d'avoir à l'initialiser à chaque fois que nous invoquons notre fonction. Après avoir initialisé votre client SDK, vous pouvez l'utiliser pour effectuer des appels d'API pour ce AWS service. L'exemple de code appelle l'action d'PutObjectAPI Amazon S3 comme suit :

```
const command = new PutObjectCommand({
    Bucket: bucketName,
    Key: key,
    Body: receiptContent
```

```
});
```

Accès aux variables d'environnement

Dans le code de votre gestionnaire, vous pouvez référencer n'importe quelle variable d'environnement à l'aide de `process.env`. Dans cet exemple, nous référençons la variable d'environnement `RECEIPT_BUCKET` définie à l'aide des lignes de code suivantes :

```
// Access environment variables
const bucketName = process.env.RECEIPT_BUCKET;
if (!bucketName) {
    throw new Error('RECEIPT_BUCKET environment variable is not set');
}
```

Utilisation de l'état global

Lambda exécute votre code statique pendant la phase d'initialisation avant d'invoquer votre fonction pour la première fois. Les ressources créées pendant l'initialisation restent en mémoire entre les invocations, ce qui vous évite d'avoir à les créer chaque fois que vous invoquez votre fonction.

Dans l'exemple de code, le code d'initialisation du client S3 se trouve en dehors du gestionnaire. L'exécution initialise le client avant que la fonction n'exécute son premier événement, et le client reste disponible pour être réutilisé pendant toutes les invocations.

Meilleures pratiques en matière de code pour les fonctions TypeScript Lambda

Suivez ces instructions pour créer des fonctions Lambda :

- Séparez le gestionnaire Lambda de votre logique principale. Cela vous permet de créer une fonction testable plus unitaire.
- Contrôlez les dépendances du package de déploiement de vos fonctions. L'environnement AWS Lambda d'exécution contient un certain nombre de bibliothèques. Pour les environnements d'exécution Node.js et Python, ceux-ci incluent le AWS SDKs. Pour activer le dernier ensemble de mises à jour des fonctionnalités et de la sécurité, Lambda met régulièrement à jour ces bibliothèques. Ces mises à jour peuvent introduire de subtiles modifications dans le comportement de votre fonction Lambda. Pour disposer du contrôle total des dépendances que votre fonction utilise, empaquetez toutes vos dépendances avec votre package de déploiement.

- Réduisez la complexité de vos dépendances. Privilégiez les infrastructures plus simples qui se chargent rapidement au démarrage de l'environnement d'exécution.
- Réduisez la taille de votre package de déploiement selon ses besoins d'exécution. Cela contribue à réduire le temps nécessaire au téléchargement et à la décompression de votre package de déploiement avant l'invocation.

Tirez parti de la réutilisation de l'environnement d'exécution pour améliorer les performances de votre fonction. Initialisez les clients SDK et les connexions à la base de données en dehors du gestionnaire de fonctions et mettez en cache les actifs statiques localement dans le répertoire `/tmp`. Les invocations ultérieures traitées par la même instance de votre fonction peuvent réutiliser ces ressources. Cela permet d'économiser des coûts, tout en réduisant le temps d'exécution de la fonction.

Pour éviter des éventuelles fuites de données entre les invocations, n'utilisez pas l'environnement d'exécution pour stocker des données utilisateur, des événements ou d'autres informations ayant un impact sur la sécurité. Si votre fonction repose sur un état réversible qui ne peut pas être stocké en mémoire dans le gestionnaire, envisagez de créer une fonction distincte ou des versions distinctes d'une fonction pour chaque utilisateur.

Utilisez une directive keep-alive pour maintenir les connexions persistantes. Lambda purge les connexions inactives au fil du temps. Si vous tentez de réutiliser une connexion inactive lorsque vous invoquez une fonction, cela entraîne une erreur de connexion. Pour maintenir votre connexion persistante, utilisez la directive Keep-alive associée à votre environnement d'exécution. Pour obtenir un exemple, consultez Réutilisation des connexions avec Keep-Alive dans Node.js.

Utilisez des variables d'environnement pour transmettre des paramètres opérationnels à votre fonction. Par exemple, si vous écrivez dans un compartiment Amazon S3 au lieu de coder en dur le nom du compartiment dans lequel vous écrivez, configurez le nom du compartiment comme variable d'environnement.

Évitez d'utiliser des invocations récursives dans votre fonction Lambda, lorsque la fonction s'invoque elle-même ou démarre un processus susceptible de l'invoquer à nouveau. Cela peut entraîner un volume involontaire d'invocations de fonction et des coûts accrus. Si vous constatez un volume involontaire d'invocations, définissez immédiatement la simultanéité réservée à la fonction sur `0` afin de limiter toutes les invocations de la fonction, pendant que vous mettez à jour le code.

N'utilisez pas de code non documenté ni public APIs dans votre code de fonction Lambda. Pour les AWS Lambda environnements d'exécution gérés, Lambda applique régulièrement des mises à jour

de sécurité et fonctionnelles aux applications internes de Lambda. APIs Ces mises à jour internes de l'API peuvent être rétroincompatibles, ce qui peut entraîner des conséquences imprévues, telles que des échecs d'invocation si votre fonction dépend de ces mises à jour non publiques. APIs Consultez la référence de l'API pour obtenir une liste des API accessibles au public APIs.

Écriture du code idempotent. L'écriture de code idempotent pour vos fonctions garantit ne gestion identique des événements dupliqués. Votre code doit valider correctement les événements et gérer correctement les événements dupliqués. Pour de plus amples informations, veuillez consulterComment faire en sorte que ma fonction Lambda soit idempotente ?.

Déployez TypeScript du code transpilé dans Lambda avec des archives de fichiers .zip

Avant de pouvoir déployer TypeScript du code sur AWS Lambda, vous devez le transpiler dans JavaScript. Cette page explique trois méthodes pour créer et déployer TypeScript du code sur Lambda avec des archives de fichiers .zip :

- En utilisant AWS Serverless Application Model (AWS SAM)
- À l'aide du AWS Cloud Development Kit (AWS CDK)
- Utilisation de la AWS Command Line Interface (AWS CLI) et d'esbuild

AWS SAM et AWS CDK simplifiez la création et le déploiement TypeScript des fonctions. La spécification du AWS SAM modèle fournit une syntaxe simple et claire pour décrire les fonctions, les autorisations APIs, les configurations et les événements Lambda qui constituent votre application sans serveur. Le AWS CDK vous permet de créer des applications fiables, évolutives et rentables dans le cloud, avec la puissance expressive considérable d'un langage de programmation. AWS CDK Il est destiné aux AWS utilisateurs modérément à très expérimentés. Le AWS CDK et le AWS SAM utilisent esbuild pour transpiler le TypeScript code dans. JavaScript

Utilisation AWS SAM pour déployer TypeScript du code sur Lambda

Suivez les étapes ci-dessous pour télécharger, créer et déployer un exemple d' TypeScript application Hello World à l'aide du AWS SAM. Cette application implémente un backend API de base. Elle se compose d'un point de terminaison Amazon API Gateway et d'une fonction Lambda. Lorsque vous envoyez une demande GET au point de terminaison API Gateway, la fonction Lambda est invoquée. La fonction renvoie un message `hello world`.

> **Note**
>
> AWS SAM utilise esbuild pour créer des fonctions Lambda Node.js à TypeScript partir du code. le support d'esbuild est actuellement en version préliminaire publique. Au cours de la version préliminaire publique, la prise en charge d'esbuild peut faire l'objet de modifications incompatibles.

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- AWS CLI version 2
- AWS SAM CLI version 1.75 ou ultérieure
- Node.js

Déployer un exemple d' AWS SAM application

1. Initialisez l'application à l'aide du TypeScript modèle Hello World.

```
sam init --app-template hello-world-typescript --name sam-app --package-type Zip --runtime nodejs24.x
```

2. (Facultatif) L'exemple d'application inclut des configurations pour les outils couramment utilisés, tels que le ESLintlinting de code et Jest pour les tests unitaires. Pour exécuter des commandes de test et de validation :

```
cd sam-app/hello-world
npm install
npm run lint
npm run test
```

3. Créez l’application.

```
cd sam-app
sam build
```

4. Déployez l’application.

```
sam deploy --guided
```

5. Suivez les invites à l’écran. Répondez avec `Enter` pour accepter les options par défaut fournies dans l’expérience interactive.
6. La sortie affiche le point de terminaison de l’API REST. Ouvrez le point de terminaison dans un navigateur pour tester la fonction. Vous devriez voir la réponse suivante :

```
{"message":"hello world"}
```

7. Il s’agit d’un point de terminaison d’API public accessible par Internet. Nous vous recommandons de supprimer le point de terminaison après un test.

```
sam delete
```

Utilisation du AWS CDK pour déployer du TypeScript code sur Lambda

Suivez les étapes ci-dessous pour créer et déployer un exemple d' TypeScript application à l'aide du AWS CDK. Cette application implémente un backend API de base. Elle se compose d’un point de terminaison API Gateway et d’une fonction Lambda. Lorsque vous envoyez une demande GET au point de terminaison API Gateway, la fonction Lambda est invoquée. La fonction renvoie un message `hello world`.

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- AWS CLI version 2
- AWS CDK version 2
- Node.js
- Docker ou esbuild

Déployer un exemple d' AWS CDK application

1. Créez un répertoire de projets pour votre nouvelle application.

```
mkdir hello-world
cd hello-world
```

2. Initialisez l’application.

```
cdk init app --language typescript
```

3. Ajoutez le pack @types/aws-lambda en tant que dépendance de développement. Ce package contient les définitions de type pour Lambda.

```
npm install -D @types/aws-lambda
```

4. Ouvrez le répertoire lib. Vous devriez voir un fichier appelé hello-world-stack.ts. Créez deux nouveaux fichiers dans ce répertoire : hello-world.function.ts et hello-world.ts.

5. Ouvrez hello-world.function.ts et ajoutez le code suivant au fichier. Il s'agit du code de la fonction Lambda.

> **Note**
>
> L'instruction `import` importe les définitions de type depuis @types /aws-lambda. Elle n'importe pas le package NPM `aws-lambda`, qui est un outil tiers indépendant. Pour plus d'informations, consultez aws-lambda dans le référentiel. DefinitelyTyped GitHub

```
import { Context, APIGatewayProxyResult, APIGatewayEvent } from 'aws-lambda';

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
    console.log(`Event: ${JSON.stringify(event, null, 2)}`);
    console.log(`Context: ${JSON.stringify(context, null, 2)}`);
    return {
        statusCode: 200,
        body: JSON.stringify({
            message: 'hello world',
        }),
    };
};
```

6. Ouvrez hello-world.ts et ajoutez le code suivant au fichier. Il contient la NodejsFunction construction, qui crée la fonction Lambda, et la LambdaRestApi construction, qui crée l'API REST.

```
import { Construct } from 'constructs';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs';
import { LambdaRestApi } from 'aws-cdk-lib/aws-apigateway';

export class HelloWorld extends Construct {
```

```
  constructor(scope: Construct, id: string) {
    super(scope, id);
    const helloFunction = new NodejsFunction(this, 'function');
    new LambdaRestApi(this, 'apigw', {
      handler: helloFunction,
    });
  }
}
```

Le composant `NodejsFunction` suppose les éléments suivants par défaut :

- Votre gestionnaire de fonctions se nomme `handler`.
- Le fichier .ts qui contient le code de fonction (hello-world.function.ts) se trouve dans le même répertoire que le fichier .ts qui contient le composant (hello-world.ts). Le composant utilise l'ID du composant (« hello-world ») et le nom du fichier du gestionnaire Lambda (« fonction ») pour trouver le code de la fonction. Par exemple, si le code de votre fonction se trouve dans un fichier nommé hello-world.my-fonction.ts, le fichier hello-world.ts doit référencer le code de la fonction comme suit :

  ```
  const helloFunction = new NodejsFunction(this, 'my-function');
  ```

Vous pouvez modifier ce comportement et configurer d'autres paramètres esbuild. Pour plus d'informations, consultez Configuration d'esbuild dans la référence de l' AWS CDK API.

7. Ouvrez hello-world-stack.ts. C'est le code qui définit votre pile de AWS CDK. Remplacez le code par ce qui suit :

```
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { HelloWorld } from './hello-world';

export class HelloWorldStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);
    new HelloWorld(this, 'hello-world');
  }
}
```

8. à partir du répertoire `hello-world` contenant votre fichier `cdk.json`, déployez votre application.

```
cdk deploy
```

9. AWS CDK Construit et empaquète la fonction Lambda à l'aide d'esbuild, puis déploie la fonction dans le moteur d'exécution Lambda. La sortie affiche le point de terminaison de l'API REST. Ouvrez le point de terminaison dans un navigateur pour tester la fonction. Vous devriez voir la réponse suivante :

   ```
   {"message":"hello world"}
   ```

   Il s'agit d'un point de terminaison d'API public accessible par Internet. Nous vous recommandons de supprimer le point de terminaison après un test.

Utilisation de AWS CLI et esbuild pour déployer TypeScript du code sur Lambda

L'exemple suivant montre comment transpiler et déployer TypeScript du code sur Lambda à l'aide d'esbuild et de. esbuild produit JavaScript un fichier avec toutes AWS CLI les dépendances. Il s'agit du seul fichier que vous devez ajouter à l'archive .zip.

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- AWS CLI version 2
- Node.js
- Un rôle d'exécution pour votre fonction Lambda
- Pour les utilisateurs de Windows, un utilitaire de fichier zip tel que 7zip.

Déployez un exemple de fonction

1. Sur votre machine locale, créez un répertoire de projets pour votre nouvelle fonction.
2. Créez un nouveau projet Node.js avec npm ou un gestionnaire de packs de votre choix.

   ```
   npm init
   ```

3. Ajoutez les packs @types/aws-lambda et esbuild en tant que dépendances de développement. Le package `@types/aws-lambda` contient les définitions de type pour Lambda.

```
npm install -D @types/aws-lambda esbuild
```

4. Créez un nouveau fichier nommé index.ts. Ajoutez le code suivant au nouveau fichier. Il s'agit du code de la fonction Lambda. La fonction renvoie un message `hello world`. La fonction ne crée aucune ressource API Gateway.

> **Note**
>
> L'instruction `import` importe les définitions de type depuis @types /aws-lambda. Elle n'importe pas le package NPM `aws-lambda`, qui est un outil tiers indépendant. Pour plus d'informations, consultez aws-lambda dans le référentiel. DefinitelyTyped GitHub

```
import { Context, APIGatewayProxyResult, APIGatewayEvent } from 'aws-lambda';

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
  console.log(`Event: ${JSON.stringify(event, null, 2)}`);
  console.log(`Context: ${JSON.stringify(context, null, 2)}`);
  return {
      statusCode: 200,
      body: JSON.stringify({
          message: 'hello world',
      }),
   };
};
```

5. Ajoutez un script de génération au fichier package.json. Ceci permet de configurer esbuild pour créer automatiquement le pack de déploiement .zip. Pour plus d'informations, consultez Scripts de génération dans la documentation d'esbuild.

   Linux and MacOS

```
"scripts": {
  "prebuild": "rm -rf dist",
  "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --
target=es2020 --outfile=dist/index.js",
```

```
  "postbuild": "cd dist && zip -r index.zip index.js*"
},
```

Windows

Dans cet exemple, la commande `"postbuild"` utilise l'utilitaire 7zip pour créer votre fichier .zip. Utilisez votre propre utilitaire Windows zip préféré et modifiez la commande si nécessaire.

```
"scripts": {
  "prebuild": "del /q dist",
  "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --
target=es2020 --outfile=dist/index.js",
  "postbuild": "cd dist && 7z a -tzip index.zip index.js*"
},
```

6. Créez le pack.

```
npm run build
```

7. Créez une fonction Lambda à l'aide du pack de déploiement .zip. Remplacez le texte surligné par l'Amazon Resource Name (ARN) de votre rôle d'exécution.

```
aws lambda create-function --function-name hello-world --runtime "nodejs24.x" --
role arn:aws:iam::123456789012:role/lambda-ex --zip-file "fileb://dist/index.zip"
 --handler index.handler
```

8. Exécutez un événement de test pour confirmer que la fonction renvoie la réponse suivante. Si vous souhaitez invoquez cette fonction à l'aide d'API Gateway, créez et configurez l'API REST.

```
{
  "statusCode": 200,
  "body": "{\"message\":\"hello world\"}"
}
```

Déployez le code TypeScript transpilé dans Lambda avec des images du conteneur

Vous pouvez déployer votre code TypeScript vers une fonction AWS Lambda en tant qu'image de conteneur Node.js. AWS fournit des images de base pour Node.js, pour vous aider à créer l'image du conteneur. Ces images de base sont préchargées avec un environnement d'exécution de langage et d'autres composants requis pour exécuter l'image sur Lambda. AWS fournit un Dockerfile pour chacune des images de base afin de vous aider à créer votre image de conteneur.

Si vous utilisez une image de base de communauté ou d'entreprise privée, vous devez y ajouter un client de l'environnement d'exécution de Node.js (RIC) pour la rendre compatible avec Lambda.

Lambda fournit un émulateur d'interface d'exécution pour les tests locaux. L'image de base AWS pour Node.js inclut l'émulateur d'interface d'exécution. Si vous utilisez une autre image de base, telle qu'une image Alpine Linux ou Debian, vous pouvez intégrer l'émulateur dans votre image ou l'installer sur votre ordinateur local.

Utilisation d'une image de base Node.js pour créer et conditionner le code de fonction TypeScript

Prérequis

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- AWS CLI version 2
- Docker (version minimale 25.0.0)
- Le plug-in buildx Docker.
- Node.js 22.x

Création d'une image à partir d'une image de base

Pour créer une image à partir d'une image de base AWS pour Lambda

1. Sur votre machine locale, créez un répertoire de projets pour votre nouvelle fonction.
2. Créez un nouveau projet Node.js avec `npm` ou un gestionnaire de packs de votre choix.

```
npm init
```

3. Ajoutez les packs @types/aws-lambda et esbuild en tant que dépendances de développement. Le package `@types/aws-lambda` contient les définitions de type pour Lambda.

```
npm install -D @types/aws-lambda esbuild
```

4. Ajoutez un script de création au fichier `package.json`.

```
  "scripts": {
  "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --
target=es2020 --outfile=dist/index.js"
}
```

5. Créez un nouveau fichier appelé `index.ts`. Ajoutez l'exemple de code suivant au nouveau fichier. Il s'agit du code de la fonction Lambda. La fonction renvoie un message `hello world`.

> **Note**
>
> L'instruction `import` importe les définitions de type depuis @types /aws-lambda. Elle n'importe pas le package NPM `aws-lambda`, qui est un outil tiers indépendant. Pour plus d'informations, voir aws-lambda dans le référentiel GitHub DefinitelyTyped.

```
import { Context, APIGatewayProxyResult, APIGatewayEvent } from 'aws-lambda';

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
    console.log(`Event: ${JSON.stringify(event, null, 2)}`);
    console.log(`Context: ${JSON.stringify(context, null, 2)}`);
    return {
        statusCode: 200,
        body: JSON.stringify({
            message: 'hello world',
        }),
    };
};
```

6. Créez un nouveau Dockerfile avec la configuration suivante :

- Définir la propriété `FROM` sur l'URI de l'image de base.
- Définissez l'argument `CMD` pour spécifier le gestionnaire de la fonction Lambda.

L'exemple de fichier Docker suivant utilise une génération en plusieurs étapes. La première étape consiste à transpirer le code TypeScript en JavaScript. La deuxième étape produit une image du conteneur contenant uniquement des fichiers JavaScript et des dépendances de production.

Notez que l'exemple de Dockerfile n'inclut pas d'instruction USER. Lorsque vous déployez une image de conteneur sur Lambda, Lambda définit automatiquement un utilisateur Linux par défaut disposant d'autorisations de moindre privilège. Ceci est différent du comportement standard de Docker qui est défini par défaut par l'utilisateur `root` lorsqu'aucune instruction USER n'est fournie.

Example Dockerfile

```
FROM public.ecr.aws/lambda/nodejs:22 as builder
WORKDIR /usr/app
COPY package.json index.ts  ./
RUN npm install
RUN npm run build

FROM public.ecr.aws/lambda/nodejs:22
WORKDIR ${LAMBDA_TASK_ROOT}
COPY --from=builder /usr/app/dist/* ./
CMD ["index.handler"]
```

7. Générez l'image Docker à l'aide de la commande docker build. L'exemple suivant nomme l'image `docker-image` et lui donne la balise `test`. Pour rendre votre image compatible avec Lambda, vous devez utiliser l'option `--provenance=false`.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> La commande spécifie l'option `--platform linux/amd64` pour garantir la compatibilité de votre conteneur avec l'environnement d'exécution Lambda, quelle que

soit l'architecture de votre machine de génération. Si vous avez l'intention de créer une fonction Lambda à l'aide de l'architecture du jeu d'instructions ARM64, veillez à modifier la commande pour utiliser plutôt l'option `--platform linux/arm64`.

(Facultatif) Testez l'image localement

1. Démarrez votre image Docker à l'aide de la commande docker run. Dans cet exemple, `docker-image` est le nom de l'image et `test` est la balise.

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

Cette commande exécute l'image en tant que conteneur et crée un point de terminaison local à `localhost:9000/2015-03-31/functions/function/invocations`.

> **Note**
>
> Si vous avez créé l'image Docker pour l'architecture du jeu d'instructions ARM64, veillez à utiliser l'option `--platform linux/arm64` plutôt que `--platform linux/amd64`.

2. À partir d'une nouvelle fenêtre de terminal, publiez un événement au point de terminaison local.

Linux/macOS

Sous Linux et macOS, exécutez la commande `curl` suivante :

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

Cette commande invoque la fonction avec un événement vide et renvoie une réponse. Si vous utilisez votre propre code de fonction plutôt que l'exemple de code de fonction, vous pouvez invoquer la fonction avec une charge utile JSON. Exemple :

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

Sous PowerShell, exécutez la commande `Invoke-WebRequest` suivante :

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Cette commande invoque la fonction avec un événement vide et renvoie une réponse. Si vous utilisez votre propre code de fonction plutôt que l'exemple de code de fonction, vous pouvez invoquer la fonction avec une charge utile JSON. Exemple :

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. Obtenez l'ID du conteneur.

```
docker ps
```

4. Utilisez la commande docker kill pour arrêter le conteneur. Dans cette commande, remplacez `3766c4ab331c` par l'ID du conteneur de l'étape précédente.

```
docker kill 3766c4ab331c
```

Déploiement de l'image

Pour charger l'image sur Amazon RIE et créer la fonction Lambda

1. Exécutez la commande get-login-password pour authentifier la CLI Docker auprès de votre registre Amazon ECR.
   - Définissez la valeur `--region` sur la Région AWS où vous voulez créer le référentiel Amazon ECR.
   - Remplacez `111122223333` par votre ID Compte AWS.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Créez un référentiel dans Amazon ECR à l'aide de la commande create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Le référentiel Amazon ECR doit être dans la même Région AWS que la fonction Lambda.

En cas de succès, vous obtenez une réponse comme celle-ci :

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Copiez le `repositoryUri` à partir de la sortie de l'étape précédente.
4. Exécutez la commande docker tag pour étiqueter votre image locale dans votre référentiel Amazon ECR en tant que dernière version. Dans cette commande :

   - `docker-image:test` est le nom et la balise de votre image Docker. Il s'agit du nom et de la balise de l'image que vous avez spécifiés dans la commande `docker build`.
   - Remplacez <ECRrepositoryUri> par l'`repositoryUri` que vous avez copié. Assurez-vous d'inclure `:latest` à la fin de l'URI.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Exemple :

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Exécutez la commande docker push pour déployer votre image locale dans le référentiel Amazon ECR. Assurez-vous d'inclure `:latest` à la fin de l'URI du référentiel.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Créez un rôle d'exécution pour la fonction, si vous n'en avez pas déjà un. Vous aurez besoin de l'Amazon Resource Name (ARN) du rôle à l'étape suivante.
7. Créez la fonction Lambda. Pour `ImageUri`, indiquez l'URI du référentiel mentionné précédemment. Assurez-vous d'inclure `:latest` à la fin de l'URI.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> Vous pouvez créer une fonction à l'aide d'une image dans un compte AWS différent, à condition que l'image soit dans la même région que la fonction Lambda. Pour de plus amples informations, consultez Autorisations entre comptes Amazon ECR.

8. Invoquer la fonction.

```
aws lambda invoke --function-name hello-world response.json
```

Vous devriez obtenir une réponse comme celle-ci :

```
{
  "ExecutedVersion": "$LATEST",
```

```
  "StatusCode": 200
}
```

9. Pour voir la sortie de la fonction, consultez le fichier `response.json`.

Pour mettre à jour le code de la fonction, vous devez recréer l'image, charger la nouvelle image dans le référentiel Amazon ECR, puis utiliser la commande update-function-code pour déployer l'image dans la fonction Lambda.

Lambda résout l'étiquette d'image en hachage d'image spécifique. Cela signifie que si vous pointez la balise d'image qui a été utilisée pour déployer la fonction vers une nouvelle image dans Amazon ECR, Lambda ne met pas automatiquement à jour la fonction pour utiliser la nouvelle image.

Pour déployer la nouvelle image sur la même fonction Lambda, vous devez utiliser la commande update-function-code, même si la balise d'image dans Amazon ECR reste la même. Dans l'exemple suivant, l'option `--publish` crée une version de la fonction à l'aide de l'image du conteneur mise à jour.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Utilisation de l'objet de contexte Lambda pour récupérer les informations relatives aux fonctions TypeScript

Lorsque Lambda exécute votre fonction, il transmet un objet contexte au gestionnaire. Cet objet fournit des méthodes et des propriétés fournissant des informations sur l'invocation, la fonction et l'environnement d'exécution.

Pour activer la vérification de type pour l'objet de contexte, vous devez ajouter le package @types/aws-lambda en tant que dépendance de développement et importer le type `Context`. Pour de plus amples informations, veuillez consulter Définitions de type pour Lambda.

Méthodes de contexte

- `getRemainingTimeInMillis()` – Renvoie le nombre de millisecondes restant avant l'expiration de l'exécution.

Propriétés du contexte

- `functionName` – Nom de la fonction Lambda.
- `functionVersion` – Version de la fonction.
- `invokedFunctionArn` – Amazon Resource Name (ARN) utilisé pour appeler la fonction. Indique si l'appelant a spécifié un numéro de version ou un alias.
- `memoryLimitInMB` – Quantité de mémoire allouée à la fonction.
- `awsRequestId` – Identifiant de la demande d'invocation.
- `logGroupName` – Groupe de journaux pour la fonction.
- `logStreamName` – Flux de journal de l'instance de fonction.
- `identity` – (applications mobiles) Informations sur l'identité Amazon Cognito qui a autorisé la demande.
  - `cognitoIdentityId` – Identité Amazon Cognito authentifiée.
  - `cognitoIdentityPoolId` – Groupe d'identités Amazon Cognito ayant autorisé l'invocation.
- `clientContext` – (applications mobiles) Contexte client fourni à Lambda par l'application client.
  - `client.installation_id`
  - `client.app_title`
  - `client.app_version_name`

- `client.app_version_code`
- `client.app_package_name`
- `env.platform_version`
- `env.platform`
- `env.make`
- `env.model`
- `env.locale`
- `Custom` – Personnalisez les valeurs qui sont définies par l'application client.

- `callbackWaitsForEmptyEventLoop`— Par défaut (`true`), lors de l'utilisation d'un gestionnaire de fonctions basé sur le rappel, Lambda attend que la boucle d'événements soit vide après l'exécution du rappel avant de mettre fin à l'appel de la fonction. Définissez cette valeur `false` sur pour envoyer la réponse et mettre fin à l'appel immédiatement après l'exécution du rappel au lieu d'attendre que la boucle d'événements soit vide. Les événements exceptionnels continuent de se dérouler lors de la prochaine invocation. Notez que Lambda prend en charge les gestionnaires de fonctions basés sur le rappel pour les environnements d'exécution de Node.js 22 et antérieurs uniquement.

Example Fichier index.ts

Dans l'exemple suivant, la fonction enregistre des informations de contexte et renvoie l'emplacement des journaux.

> **Note**
>
> Avant d'utiliser ce code dans une fonction Lambda, vous devez ajouter le package @types/aws-lambda en tant que dépendance de développement. Ce package contient les définitions de type pour Lambda. Pour de plus amples informations, veuillez consulter Définitions de type pour Lambda.

```
import { Context } from 'aws-lambda';
export const lambdaHandler = async (event: string, context: Context): Promise<string>
 => {
  console.log('Remaining time: ', context.getRemainingTimeInMillis());
  console.log('Function name: ', context.functionName);
  return context.logStreamName;
```

```
};
```

Enregistrez et surveillez les fonctions TypeScript Lambda

AWS Lambda surveille automatiquement les fonctions Lambda et envoie des entrées de journal à Amazon. CloudWatch Votre fonction Lambda est fournie avec un groupe de CloudWatch journaux Logs et un flux de journaux pour chaque instance de votre fonction. L'environnement d'exécution Lambda envoie des détails sur chaque invocation et d'autres sorties provenant du code de votre fonction au flux de journaux. Pour plus d'informations sur CloudWatch les journaux, consultezEnvoi automatique des journaux de la fonction Lambda à CloudWatch Logs.

Pour générer les journaux à partir de votre code de fonction, vous pouvez utiliser des méthodes sur l'objet console. Pour une journalisation plus détaillée, vous pouvez utiliser n'importe quelle bibliothèque de journalisation qui écrit dans `stdout` ou `stderr`.

Sections

- Utilisation d'outils et de bibliothèques de journalisation
- Utilisation de Powertools pour AWS Lambda (TypeScript) et AWS SAM pour la journalisation structurée
- Utilisation de Powertools pour AWS Lambda (TypeScript) et AWS CDK pour la journalisation structurée
- Affichage des journaux dans la console Lambda
- Afficher les journaux dans la CloudWatch console

Utilisation d'outils et de bibliothèques de journalisation

Powertools for AWS Lambda (TypeScript) est une boîte à outils destinée aux développeurs qui permet de mettre en œuvre les meilleures pratiques sans serveur et d'accroître la rapidité des développeurs. L'utilitaire Logger fournit un enregistreur optimisé pour Lambda qui inclut des informations supplémentaires sur le contexte de fonction pour toutes vos fonctions avec une sortie structurée en JSON. Utilisez cet utilitaire pour effectuer les opérations suivantes :

- Capturer les champs clés du contexte Lambda, démarrer à froid et structurer la sortie de la journalisation sous forme de JSON
- Journaliser les événements d'invocation Lambda lorsque cela est demandé (désactivé par défaut)
- Imprimer tous les journaux uniquement pour un pourcentage d'invocations via l'échantillonnage des journaux (désactivé par défaut)
- Ajouter des clés supplémentaires au journal structuré à tout moment

- Utiliser un formateur de journaux personnalisé (Apportez votre propre formateur) pour produire des journaux dans une structure compatible avec le RFC de journalisation de votre organisation

Utilisation de Powertools pour AWS Lambda (TypeScript) et AWS SAM pour la journalisation structurée

Suivez les étapes ci-dessous pour télécharger, créer et déployer un exemple d' TypeScript application Hello World avec des modules Powertools for AWS Lambda (TypeScript) intégrés à l'aide du AWS SAM. Cette application met en œuvre un backend API de base et utilise Powertools pour émettre des journaux, des métriques et des traces. Elle se compose d'un point de terminaison Amazon API Gateway et d'une fonction Lambda. Lorsque vous envoyez une requête GET au point de terminaison API Gateway, la fonction Lambda appelle, envoie des journaux et des métriques au format métrique intégré à CloudWatch, et envoie des traces à. AWS X-Ray La fonction renvoie un message `hello world`.

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- Node.js 20 ou version ultérieure
- AWS CLI version 2
- AWS SAM CLI version 1.75 ou ultérieure. Si vous disposez d'une ancienne version de la AWS SAM CLI, reportez-vous à la section Mise à niveau de la AWS SAM CLI.

Déployer un exemple d' AWS SAM application

1. Initialisez l'application à l'aide du TypeScript modèle Hello World.

```
sam init --app-template hello-world-powertools-typescript --name sam-app --package-type Zip --runtime nodejs24.x
```

2. Créez l'application.

```
cd sam-app && sam build
```

3. Déployez l'application.

```
sam deploy --guided
```

4. Suivez les invites à l'écran. Appuyez sur `Enter` pour accepter les options par défaut fournies dans l'expérience interactive.

> **Note**
>
> Car l'autorisation n'a HelloWorldFunction peut-être pas été définie, est-ce que ça va ? , assurez-vous de participery.

5. Obtenez l'URL de l'application déployée :

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoquez le point de terminaison de l'API :

```
curl <URL_FROM_PREVIOUS_STEP>
```

En cas de succès, vous obtiendrez cette réponse :

```
{"message":"hello world"}
```

7. Pour obtenir les journaux de la fonction, exécutez sam logs. Pour en savoir plus, consultez Utilisation des journaux dans le Guide du développeur AWS Serverless Application Model .

```
sam logs --stack-name sam-app
```

La sortie du journal se présente comme suit :

```
2025/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2025-08-31T09:33:10.552000
 START RequestId: 70693159-7e94-4102-a2af-98a6343fb8fb Version: $LATEST
2025/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2025-08-31T09:33:10.594000
 2025-08-31T09:33:10.557Z 70693159-7e94-4102-a2af-98a6343fb8fb
 INFO {"_aws":{"Timestamp":1661938390556,"CloudWatchMetrics":
[{"Namespace":"sam-app","Dimensions":[["service"]],"Metrics":
[{"Name":"ColdStart","Unit":"Count"}]}]},"service":"helloWorld","ColdStart":1}
2025/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2025-08-31T09:33:10.595000
 2025-08-31T09:33:10.595Z 70693159-7e94-4102-a2af-98a6343fb8fb INFO
```

```
 {"level":"INFO","message":"This is an INFO log - sending HTTP 200 - hello world
 response","service":"helloWorld","timestamp":"2025-08-31T09:33:10.594Z"}
2025/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2025-08-31T09:33:10.655000
 2025-08-31T09:33:10.655Z 70693159-7e94-4102-a2af-98a6343fb8fb INFO
 {"_aws":{"Timestamp":1661938390655,"CloudWatchMetrics":[{"Namespace":"sam-
app","Dimensions":[["service"]],"Metrics":[]}]},"service":"helloWorld"}
2025/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2025-08-31T09:33:10.754000 END
 RequestId: 70693159-7e94-4102-a2af-98a6343fb8fb
2025/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2025-08-31T09:33:10.754000
 REPORT RequestId: 70693159-7e94-4102-a2af-98a6343fb8fb Duration: 201.55 ms Billed
 Duration: 202 ms Memory Size: 128 MB Max Memory Used: 66 MB Init Duration: 252.42
 ms
XRAY TraceId: 1-630f2ad5-1de22b6d29a658a466e7ecf5 SegmentId: 567c116658fbf11a
 Sampled: true
```

8. Il s'agit d'un point de terminaison d'API public accessible par Internet. Nous vous recommandons de supprimer le point de terminaison après un test.

```
sam delete
```

Gestion de la conservation des journaux

Les groupes de journaux ne sont pas supprimés automatiquement quand vous supprimez une fonction. Pour éviter de stocker les journaux indéfiniment, supprimez le groupe de journaux ou configurez une période de conservation après laquelle les journaux CloudWatch sont automatiquement supprimés. Pour configurer la conservation des journaux, ajoutez ce qui suit à votre AWS SAM modèle :

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
    # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Utilisation de Powertools pour AWS Lambda (TypeScript) et AWS CDK pour la journalisation structurée

Suivez les étapes ci-dessous pour télécharger, créer et déployer un exemple d' TypeScript application Hello World avec des modules Powertools for AWS Lambda (TypeScript) intégrés à l'aide du AWS CDK. Cette application met en œuvre un backend API de base et utilise Powertools pour émettre des journaux, des métriques et des traces. Elle se compose d'un point de terminaison Amazon API Gateway et d'une fonction Lambda. Lorsque vous envoyez une requête GET au point de terminaison API Gateway, la fonction Lambda appelle, envoie des journaux et des métriques au format métrique intégré à CloudWatch, et envoie des traces à. AWS X-Ray La fonction renvoie un message `hello world`.

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- Node.js 20 ou version ultérieure
- AWS CLI version 2
- AWS CDK version 2
- AWS SAM CLI version 1.75 ou ultérieure. Si vous disposez d'une ancienne version de la AWS SAM CLI, reportez-vous à la section Mise à niveau de la AWS SAM CLI.

Déployer un exemple d' AWS CDK application

1. Créez un répertoire de projets pour votre nouvelle application.

```
mkdir hello-world
cd hello-world
```

2. Initialisez l'application.

```
cdk init app --language typescript
```

3. Ajoutez le pack @types/aws-lambda en tant que dépendance de développement.

```
npm install -D @types/aws-lambda
```

4. Installez l'utilitaire Powertools Logger (français non garanti).

```
npm install @aws-lambda-powertools/logger
```

5. Ouvrez le répertoire lib. Vous devriez voir un fichier appelé hello-world-stack.ts. Créez deux nouveaux fichiers dans ce répertoire : hello-world.function.ts et hello-world.ts.
6. Ouvrez hello-world.function.ts et ajoutez le code suivant au fichier. Il s'agit du code de la fonction Lambda.

```
import { APIGatewayEvent, APIGatewayProxyResult, Context } from 'aws-lambda';
import { Logger } from '@aws-lambda-powertools/logger';
const logger = new Logger();

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
  logger.info('This is an INFO log - sending HTTP 200 - hello world response');
  return {
    statusCode: 200,
    body: JSON.stringify({
      message: 'hello world',
    }),
  };
};
```

7. Ouvrez hello-world.ts et ajoutez le code suivant au fichier. Il contient la NodejsFunction construction qui crée la fonction Lambda, configure les variables d'environnement pour Powertools et définit la durée de conservation des journaux à une semaine. Il inclut également la LambdaRestApi construction qui crée l'API REST.

```
import { Construct } from 'constructs';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs';
import { LambdaRestApi } from 'aws-cdk-lib/aws-apigateway';
import { RetentionDays } from 'aws-cdk-lib/aws-logs';
import { CfnOutput } from 'aws-cdk-lib';

export class HelloWorld extends Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);
    const helloFunction = new NodejsFunction(this, 'function', {
      environment: {
        Powertools_SERVICE_NAME: 'helloWorld',
        LOG_LEVEL: 'INFO',
      },
```

```
      logRetention: RetentionDays.ONE_WEEK,
    });
    const api = new LambdaRestApi(this, 'apigw', {
      handler: helloFunction,
    });
    new CfnOutput(this, 'apiUrl', {
      exportName: 'apiUrl',
      value: api.url,
    });
  }
}
```

8. Ouvrez hello-world-stack.ts. C'est le code qui définit votre pile de AWS CDK. Remplacez le code par ce qui suit :

```
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { HelloWorld } from './hello-world';

export class HelloWorldStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);
    new HelloWorld(this, 'hello-world');
  }
}
```

9. Revenez au répertoire du projet.

```
cd hello-world
```

10. Déployez votre application.

```
cdk deploy
```

11. Obtenez l'URL de l'application déployée :

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?ExportName==`apiUrl`].OutputValue' --output text
```

12. Invoquez le point de terminaison de l'API :

```
curl <URL_FROM_PREVIOUS_STEP>
```

En cas de succès, vous obtiendrez cette réponse :

```
{"message":"hello world"}
```

13. Pour obtenir les journaux de la fonction, exécutez sam logs. Pour en savoir plus, consultez Utilisation des journaux dans le Guide du développeur AWS Serverless Application Model .

```
sam logs --stack-name HelloWorldStack
```

La sortie du journal se présente comme suit :

```
2025/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2025-08-31T14:48:37.047000
 START RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c Version: $LATEST
2025/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2025-08-31T14:48:37.050000 {
"level": "INFO",
"message": "This is an INFO log - sending HTTP 200 - hello world response",
"service": "helloWorld",
"timestamp": "2025-08-31T14:48:37.048Z",
"xray_trace_id": "1-630f74c4-2b080cf77680a04f2362bcf2"
}
2025/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2025-08-31T14:48:37.082000 END
 RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c
2025/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2025-08-31T14:48:37.082000
 REPORT RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c Duration: 34.60 ms Billed
 Duration: 35 ms Memory Size: 128 MB Max Memory Used: 57 MB Init Duration: 173.48
 ms
```

14. Il s’agit d’un point de terminaison d’API public accessible par Internet. Nous vous recommandons de supprimer le point de terminaison après un test.

```
cdk destroy
```

Affichage des journaux dans la console Lambda

Vous pouvez utiliser la console Lambda pour afficher la sortie du journal après avoir invoqué une fonction Lambda.

Si votre code peut être testé à partir de l'éditeur Code intégré, vous trouverez les journaux dans les résultats d'exécution. Lorsque vous utilisez la fonctionnalité de test de console pour invoquer une fonction, vous trouverez Sortie du journal dans la section Détails.

Afficher les journaux dans la CloudWatch console

Vous pouvez utiliser la CloudWatch console Amazon pour consulter les journaux de toutes les invocations de fonctions Lambda.

Pour afficher les journaux sur la CloudWatch console

1. Ouvrez la page Groupes de journaux sur la CloudWatch console.
2. Choisissez le groupe de journaux pour votre fonction (***your-function-name***/aws/lambda/).
3. Choisissez un flux de journaux.

Chaque flux de journal correspond à une instance de votre fonction. Un flux de journal apparaît lorsque vous mettez à jour votre fonction Lambda et lorsque des instances supplémentaires sont créées pour gérer les appels simultanés. Pour trouver les journaux d'un appel spécifique, nous vous recommandons d'instrumenter votre fonction avec. AWS X-Ray X-Ray enregistre des détails sur la demande et le flux de journaux dans le suivi.

TypeScript Code de suivi dans AWS Lambda

Lambda s'intègre pour vous aider AWS X-Ray à suivre, à déboguer et à optimiser les applications Lambda. Vous pouvez utiliser X-Ray pour suivre une demande lorsque celle-ci parcourt les ressources de votre application, qui peuvent inclure des fonctions Lambda et d'autres services AWS .

Pour envoyer des données de traçage à X-Ray, vous pouvez utiliser l'une des trois bibliothèques SDK suivantes :

- AWS Distro for OpenTelemetry (ADOT) — Une distribution sécurisée, prête pour la production et AWS prise en charge du SDK (). OpenTelemetry OTel
- AWS X-Ray SDK pour Node.js : SDK permettant de générer et d'envoyer des données de suivi à X-Ray.
- Powertools for AWS Lambda (TypeScript) — Une boîte à outils pour les développeurs permettant de mettre en œuvre les meilleures pratiques sans serveur et d'accroître la rapidité des développeurs.

Chacune d'entre elles SDKs propose des moyens d'envoyer vos données de télémétrie au service X-Ray. Vous pouvez ensuite utiliser X-Ray pour afficher, filtrer et avoir un aperçu des métriques de performance de votre application, afin d'identifier les problèmes et les occasions d'optimiser votre application.

> **Important**
>
> Les outils X-Ray et Powertools pour AWS Lambda SDKs font partie d'une solution d'instrumentation étroitement intégrée proposée par AWS. Les couches ADOT Lambda font partie d'une norme industrielle pour l'instrumentation de traçage qui collecte plus de données en général, mais qui peut ne pas convenir à tous les cas d'utilisation. Vous pouvez implémenter le end-to-end traçage dans X-Ray en utilisant l'une ou l'autre solution. Pour en savoir plus sur le choix entre les deux, consultez Choosing between the AWS Distro for Open Telemetry and X-Ray. SDKs

Sections

- Utilisation de Powertools pour AWS Lambda (TypeScript) et AWS SAM pour le traçage
- Utilisation de Powertools pour AWS Lambda (TypeScript) et AWS CDK pour le traçage
- Interprétation d'un suivi X-Ray

Utilisation de Powertools pour AWS Lambda (TypeScript) et AWS SAM pour le traçage

Suivez les étapes ci-dessous pour télécharger, créer et déployer un exemple d' TypeScript application Hello World avec des modules Powertools for AWS Lambda (TypeScript) intégrés à l'aide du AWS SAM. Cette application met en œuvre un backend API de base et utilise Powertools pour émettre des journaux, des métriques et des traces. Elle se compose d'un point de terminaison Amazon API Gateway et d'une fonction Lambda. Lorsque vous envoyez une requête GET au point de terminaison API Gateway, la fonction Lambda appelle, envoie des journaux et des métriques au format métrique intégré à CloudWatch, et envoie des traces à. AWS X-Ray La fonction renvoie un message `hello world`.

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- Node.js
- AWS CLI version 2
- AWS SAM CLI version 1.75 ou ultérieure. Si vous disposez d'une ancienne version de la AWS SAM CLI, reportez-vous à la section Mise à niveau de la AWS SAM CLI.

Déployer un exemple d' AWS SAM application

1. Initialisez l'application à l'aide du TypeScript modèle Hello World.

```
sam init --app-template hello-world-powertools-typescript --name sam-app --package-type Zip --runtime nodejs24.x --no-tracing
```

2. Créez l'application.

```
cd sam-app && sam build
```

3. Déployez l'application.

```
sam deploy --guided
```

4. Suivez les invites à l'écran. Appuyez sur `Enter` pour accepter les options par défaut fournies dans l'expérience interactive.

> **Note**
>
> Car l'autorisation n'a HelloWorldFunction peut-être pas été définie, est-ce que ça va ? , assurez-vous de participery.

5. Obtenez l'URL de l'application déployée :

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoquez le point de terminaison de l'API :

```
curl <URL_FROM_PREVIOUS_STEP>
```

En cas de succès, vous obtiendrez cette réponse :

```
{"message":"hello world"}
```

7. Pour obtenir les traces de la fonction, exécutez sam traces.

```
sam traces
```

La sortie de la trace ressemble à ceci :

```
XRay Event [revision 1] at (2023-01-31T11:29:40.527000) with id
 (1-11a2222-111a222222cb33de3b95daf9) and duration (0.483s)
  - 0.425s - sam-app/Prod [HTTP: 200]
    - 0.422s - Lambda [HTTP: 200]
  - 0.406s - sam-app-HelloWorldFunction-Xyzv11a1bcde [HTTP: 200]
  - 0.172s - sam-app-HelloWorldFunction-Xyzv11a1bcde
    - 0.179s - Initialization
    - 0.112s - Invocation
      - 0.052s - ## app.lambdaHandler
        - 0.001s - ### MySubSegment
    - 0.059s - Overhead
```

8. Il s'agit d'un point de terminaison d'API public accessible par Internet. Nous vous recommandons de supprimer le point de terminaison après un test.

```
sam delete
```

X-Ray ne trace pas toutes les requêtes vers votre application. X-Ray applique un algorithme d'échantillonnage pour s'assurer que le suivi est efficace, tout en fournissant un échantillon représentatif de toutes les demandes. Le taux d'échantillonnage est 1 demande par seconde et 5 % de demandes supplémentaires. Vous ne pouvez pas configurer ce taux d'échantillonnage X-Ray pour vos fonctions.

Utilisation de Powertools pour AWS Lambda (TypeScript) et AWS CDK pour le traçage

Suivez les étapes ci-dessous pour télécharger, créer et déployer un exemple d' TypeScript application Hello World avec des modules Powertools for AWS Lambda (TypeScript) intégrés à l'aide du AWS CDK. Cette application met en œuvre un backend API de base et utilise Powertools pour émettre des journaux, des métriques et des traces. Elle se compose d'un point de terminaison Amazon API Gateway et d'une fonction Lambda. Lorsque vous envoyez une requête GET au point de terminaison API Gateway, la fonction Lambda appelle, envoie des journaux et des métriques au format métrique intégré à CloudWatch, et envoie des traces à. AWS X-Ray La fonction renvoie un message `hello world`.

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- Node.js
- AWS CLI version 2
- AWS CDK version 2
- AWS SAM CLI version 1.75 ou ultérieure. Si vous disposez d'une ancienne version de la AWS SAM CLI, reportez-vous à la section Mise à niveau de la AWS SAM CLI.

Déployer un exemple d' AWS Cloud Development Kit (AWS CDK) application

1. Créez un répertoire de projets pour votre nouvelle application.

```
mkdir hello-world
```

```
cd hello-world
```

2. Initialisez l'application.

```
cdk init app --language typescript
```

3. Ajoutez le pack @types/aws-lambda en tant que dépendance de développement.

```
npm install -D @types/aws-lambda
```

4. Installez l'utilitaire Powertools Tracer.

```
npm install @aws-lambda-powertools/tracer
```

5. Ouvrez le répertoire lib. Vous devriez voir un fichier appelé hello-world-stack.ts. Créez deux nouveaux fichiers dans ce répertoire : hello-world.function.ts et hello-world.ts.

6. Ouvrez hello-world.function.ts et ajoutez le code suivant au fichier. Il s'agit du code de la fonction Lambda.

```
import { APIGatewayEvent, APIGatewayProxyResult, Context } from 'aws-lambda';
import { Tracer } from '@aws-lambda-powertools/tracer';
const tracer = new Tracer();

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
  // Get facade segment created by Lambda
  const segment = tracer.getSegment();

  // Create subsegment for the function and set it as active
  const handlerSegment = segment.addNewSubsegment(`## ${process.env._HANDLER}`);
  tracer.setSegment(handlerSegment);

  // Annotate the subsegment with the cold start and serviceName
  tracer.annotateColdStart();
  tracer.addServiceNameAnnotation();

  // Add annotation for the awsRequestId
  tracer.putAnnotation('awsRequestId', context.awsRequestId);
  // Create another subsegment and set it as active
  const subsegment = handlerSegment.addNewSubsegment('### MySubSegment');
  tracer.setSegment(subsegment);
  let response: APIGatewayProxyResult = {
```

```
    statusCode: 200,
    body: JSON.stringify({
      message: 'hello world',
    }),
  };
  // Close subsegments (the Lambda one is closed automatically)
  subsegment.close(); // (### MySubSegment)
  handlerSegment.close(); // (## index.handler)

  // Set the facade segment as active again (the one created by Lambda)
  tracer.setSegment(segment);
  return response;
};
```

7. Ouvrez hello-world.ts et ajoutez le code suivant au fichier. Il contient la NodejsFunction construction qui crée la fonction Lambda, configure les variables d'environnement pour Powertools et définit la durée de conservation des journaux à une semaine. Il inclut également la LambdaRestApi construction qui crée l'API REST.

```
import { Construct } from 'constructs';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs';
import { LambdaRestApi } from 'aws-cdk-lib/aws-apigateway';
import { CfnOutput } from 'aws-cdk-lib';
import { Tracing } from 'aws-cdk-lib/aws-lambda';

export class HelloWorld extends Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);
    const helloFunction = new NodejsFunction(this, 'function', {
      environment: {
        POWERTOOLS_SERVICE_NAME: 'helloWorld',
      },
      tracing: Tracing.ACTIVE,
    });
    const api = new LambdaRestApi(this, 'apigw', {
      handler: helloFunction,
    });
    new CfnOutput(this, 'apiUrl', {
      exportName: 'apiUrl',
      value: api.url,
    });
  }
}
```

8. Ouvrez hello-world-stack.ts. C'est le code qui définit votre pile de AWS CDK. Remplacez le code par ce qui suit :

```
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { HelloWorld } from './hello-world';

export class HelloWorldStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);
    new HelloWorld(this, 'hello-world');
  }
}
```

9. Déployez votre application.

```
cd ..
cdk deploy
```

10. Obtenez l'URL de l'application déployée :

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?ExportName==`apiUrl`].OutputValue' --output text
```

11. Invoquez le point de terminaison de l'API :

```
curl <URL_FROM_PREVIOUS_STEP>
```

En cas de succès, vous obtiendrez cette réponse :

```
{"message":"hello world"}
```

12. Pour obtenir les traces de la fonction, exécutez sam traces.

```
sam traces
```

La sortie de la trace ressemble à ceci :

```
XRay Event [revision 1] at (2023-01-31T11:50:06.997000) with id
 (1-11a2222-111a222222cb33de3b95daf9) and duration (0.449s)
  - 0.350s - HelloWorldStack-helloworldfunction111A2BCD-Xyzv11a1bcde [HTTP: 200]
```

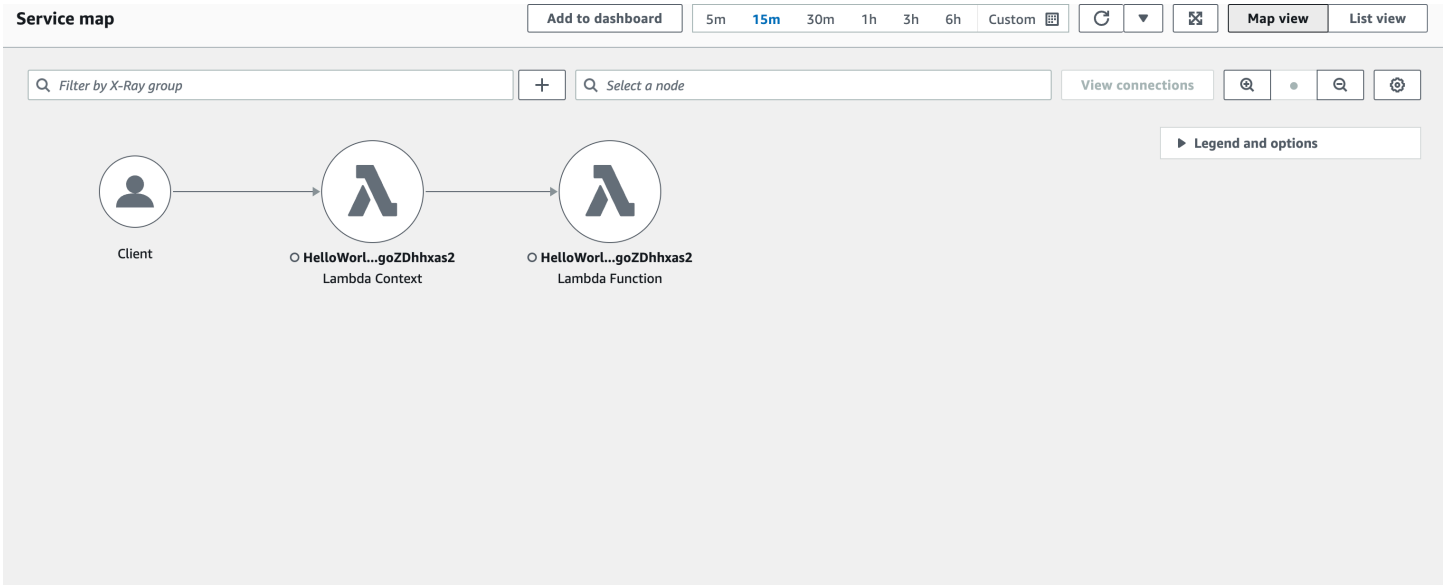
```
  - 0.157s - HelloWorldStack-helloworldfunction111A2BCD-Xyzv11a1bcde
    - 0.169s - Initialization
    - 0.058s - Invocation
      - 0.055s - ## index.handler
        - 0.000s - ### MySubSegment
    - 0.099s - Overhead
```

13. Il s'agit d'un point de terminaison d'API public accessible par Internet. Nous vous recommandons de supprimer le point de terminaison après un test.

```
cdk destroy
```

Interprétation d'un suivi X-Ray

Une fois que vous avez configuré le suivi actif, vous pouvez observer des demandes spécifiques via votre application. La mappe de trace X-Ray fournit des informations sur votre application et tous ses composants. L'exemple suivant montre une trace à partir de l'exemple d'application :



Création de fonctions Lambda avec Python

Vous pouvez exécuter du code Python dans AWS Lambda. Lambda fournit des runtimes pour Python qui exécutent votre code afin de traiter des événements. Votre code s'exécute dans un environnement qui inclut le SDK pour Python (Boto3), avec les informations d'identification d'un rôle (IAM) que vous Gestion des identités et des accès AWS gérez. Pour en savoir plus sur les versions du kit SDK incluses dans les environnements d'exécution Python, consultez the section called "Versions du SDK incluses dans l'environnement d'exécution".

Lambda prend en charge les runtimes Python suivants.

Name	Identifiant	Système d'exploitation	Date d'obsolescence	Créer la fonction de blocage	Mettre à jour la fonction de blocage
Python 3.14	`python3.14`	Amazon Linux 2	30 juin 2029	31 juillet 2029	31 août 2029
Python 3.13	`python3.13`	Amazon Linux 2	30 juin 2029	31 juillet 2029	31 août 2029
Python 3.12	`python3.12`	Amazon Linux 2	31 octobre 2028	30 novembre 2028	10 janvier 2029
Python 3.11	`python3.11`	Amazon Linux 2	30 juin 2027	31 juillet 2027	31 août 2027
Python 3.10	`python3.10`	Amazon Linux 2	31 octobre 2026	30 novembre 2026	15 janvier 2027

Pour créer une fonction Python

1. Ouvrez la console Lambda.
2. Sélectionnez Créer une fonction.
3. Configurez les paramètres suivants :
   - Nom de la fonction : saisissez le nom de la fonction.

- Runtime : choisissez Python 3.14.

4. Choisissez Créer une fonction.

La console crée une fonction Lambda avec un seul fichier source nommé `lambda_function`. Vous pouvez modifier ce fichier et ajouter d'autres fichiers dans l'éditeur de code intégré. Dans la section DÉPLOYER, choisissez Déployer pour mettre à jour le code de votre fonction. Ensuite, pour exécuter votre code, choisissez Créer un événement de test dans la section ÉVÉNEMENTS DE TEST.

Votre fonction Lambda est fournie avec un groupe de CloudWatch journaux Logs. La fonction runtime envoie les détails de chaque appel à CloudWatch Logs. Il relaie tous les journaux que votre fonction génère pendant l'invocation. Si votre fonction renvoie une erreur, Lambda met en forme l'erreur et la renvoie à l'appelant.

Rubriques

- Versions du SDK incluses dans l'environnement d'exécution
- Fonctionnalités Python désactivées
- Format de la réponse
- Arrêt progressif pour les extensions
- Définition du gestionnaire de fonction Lambda en Python
- Travailler avec des archives de fichiers .zip pour les fonctions Lambda Python
- Déployer des fonctions Lambda en Python avec des images conteneurs
- Utilisation de couches pour les fonctions Lambda Python
- Utilisation de l'objet de contexte Lambda pour récupérer les informations de la fonction Python
- Journalisation et surveillance des fonctions Lambda Python
- Test de la fonction AWS Lambda dans Python
- Instrumentation du code Python dans AWS Lambda

Versions du SDK incluses dans l'environnement d'exécution

La version du AWS SDK incluse dans le runtime Python dépend de la version d'exécution et de votre Région AWS. Pour trouver la version du kit SDK incluse dans l'environnement d'exécution que vous utilisez, créez une fonction Lambda avec le code suivant.

```
import boto3
```

```
import botocore

def lambda_handler(event, context):
   print(f'boto3 version: {boto3.__version__}')
   print(f'botocore version: {botocore.__version__}')
```

Fonctionnalités Python désactivées

Le tableau suivant répertorie les fonctionnalités Python qui sont désactivées dans les environnements d'exécution gérés par Lambda et les images de base de conteneurs pour Python. Ces fonctionnalités doivent être activées lors de la compilation de l'exécutable Python et ne peuvent pas être activées à l'aide d'un indicateur d'exécution. Pour utiliser ces fonctionnalités dans Lambda, vous pouvez déployer votre propre version d'exécution Python avec ces fonctionnalités activées, à l'aide d'une image de conteneur ou d'un environnement d'exécution personnalisé.

Fonctionnalité Python	Versions de Python concernées	Statut
Just-in-Time compilateur (JIT)	Python 3.13 et versions ultérieures	Le compilateur JIT est expérimental et n'est pas recommandé pour les charges de travail de production. Il est donc désactivé dans les environnements d'exécution Lambda Python.
Filetage gratuit	Python 3.13 et versions ultérieures	Le threading gratuit (option permettant de désactiver le verrouillage global de l'interpr éteur) est désactivé dans les versions Lambda Python en raison de l'impact sur les performan ces du code à thread unique.

Format de la réponse

Dans les environnements d'exécution Python 3.12 et versions ultérieures, les fonctions renvoient des caractères Unicode dans le cadre de leur réponse JSON. Les environnements d'exécution Python antérieurs renvoyaient des séquences échappées pour les caractères Unicode dans les réponses. Par exemple, dans Python 3.11, si vous renvoyez une chaîne Unicode telle que « こんにちは », elle échappe le caractères Unicode et renvoie « \u3053\u3093\u306b\u3061\u306f ». L'environnement d'exécution Python 3.12 renvoie le « こんにちは » d'origine.

L'utilisation de réponses Unicode réduit la taille des réponses Lambda, ce qui facilite l'adaptation de réponses plus importantes à la taille de charge utile maximale de 6 Mo pour les fonctions synchrones. Dans l'exemple précédent, la version échappée est de 32 octets, contre 17 octets pour la chaîne Unicode.

Lors de la mise à niveau vers Python 3.12 ou à une version ultérieure de l'environnement d'exécution Python, vous devrez peut-être ajuster votre code pour tenir compte du nouveau format de réponse. Si l'appelant s'attend à une chaîne Unicode échappée, vous devez soit ajouter du code à la fonction de renvoi pour échapper l'Unicode manuellement, soit ajuster l'appelant pour qu'il gère le retour Unicode.

Arrêt progressif pour les extensions

Les environnements d'exécution Python 3.12 et versions ultérieures offrent des fonctionnalités d'arrêt progressif améliorées pour les fonctions dotées d'extensions externes. Quand Lambda arrête l'exécution, il envoie un signal `SIGTERM` à l'exécution, puis un événement `SHUTDOWN` à chaque extension externe enregistrée. Vous pouvez intercepter le signal `SIGTERM` dans votre fonction Lambda et nettoyer les ressources telles que les connexions de base de données créées par la fonction.

Pour en savoir plus sur le cycle de vie de l'environnement d'exécution, veuillez consulter Comprendre le cycle de vie de l'environnement d'exécution Lambda. Pour des exemples d'utilisation de l'arrêt progressif avec des extensions, consultez le GitHub référentiel AWS Samples.

Définition du gestionnaire de fonction Lambda en Python

Le gestionnaire de fonction Lambda est la méthode dans votre code de fonction qui traite les événements. Lorsque votre fonction est invoquée, Lambda exécute la méthode du gestionnaire. Votre fonction s'exécute jusqu'à ce que le gestionnaire renvoie une réponse, se ferme ou expire.

Cette page explique comment utiliser les gestionnaires de fonctions Lambda dans Python, notamment les conventions de dénomination, les signatures de gestionnaire valides et les bonnes pratiques. Cette page inclut également un exemple de fonction Lambda Python qui collecte des informations relatives à une commande, produit un reçu sous forme de fichier texte et place ce fichier dans un compartiment Amazon Simple Storage Service (Amazon S3).

Rubriques

- Exemple de fonction Lambda Python
- Convention de nommage du gestionnaire
- Utilisation de l'objet d'événement Lambda
- Accès et utilisation de l'objet de contexte Lambda
- Signatures de gestionnaire valides pour les gestionnaires Python
- Renvoi d'une valeur
- Utilisation d'AWS SDK pour Python (Boto3) dans votre gestionnaire
- Accès aux variables d'environnement
- Pratiques exemplaires en matière de code pour les fonctions Lambda Python

Exemple de fonction Lambda Python

L'exemple de code de fonction Lambda Python suivant prend en compte les informations relatives à une commande, produit un reçu sous forme de fichier texte et place ce fichier dans un compartiment Amazon S3 :

Example Fonction Lambda Python

```
import json
import os
import logging
import boto3

# Initialize the S3 client outside of the handler
```

```
s3_client = boto3.client('s3')

# Initialize the logger
logger = logging.getLogger()
logger.setLevel("INFO")

def upload_receipt_to_s3(bucket_name, key, receipt_content):
    """Helper function to upload receipt to S3"""

    try:
        s3_client.put_object(
            Bucket=bucket_name,
            Key=key,
            Body=receipt_content
        )
    except Exception as e:
        logger.error(f"Failed to upload receipt to S3: {str(e)}")
        raise

def lambda_handler(event, context):
    """
    Main Lambda handler function
    Parameters:
        event: Dict containing the Lambda function event data
        context: Lambda runtime context
    Returns:
        Dict containing status message
    """
    try:
        # Parse the input event
        order_id = event['Order_id']
        amount = event['Amount']
        item = event['Item']

        # Access environment variables
        bucket_name = os.environ.get('RECEIPT_BUCKET')
        if not bucket_name:
            raise ValueError("Missing required environment variable RECEIPT_BUCKET")

        # Create the receipt content and key destination
        receipt_content = (
            f"OrderID: {order_id}\n"
            f"Amount: ${amount}\n"
            f"Item: {item}"
```

```
        )
        key = f"receipts/{order_id}.txt"

        # Upload the receipt to S3
        upload_receipt_to_s3(bucket_name, key, receipt_content)

        logger.info(f"Successfully processed order {order_id} and stored receipt in S3
bucket {bucket_name}")

        return {
            "statusCode": 200,
            "message": "Receipt processed successfully"
        }

    except Exception as e:
        logger.error(f"Error processing order: {str(e)}")
        raise
```

Ce fichier comprend les sections suivantes :

- Bloc `import` : utilisez ce bloc pour inclure les bibliothèques que votre fonction Lambda requiert.
- Initialisation globale du client SDK et de l'enregistreur : l'inclusion du code d'initialisation en dehors du gestionnaire permet de tirer parti de la réutilisation de l'environnement d'exécution afin d'améliorer les performances de votre fonction. Pour en savoir plus, veuillez consulter the section called "Pratiques exemplaires en matière de code pour les fonctions Lambda Python".
- `def upload_receipt_to_s3(bucket_name, key, receipt_content):` une fonction d'assistance appelée par la fonction `lambda_handler` principale.
- `def lambda_handler(event, context):` la fonction de gestion principale de votre code, qui contient la logique principale de votre application. Lorsque Lambda invoque votre gestionnaire de fonction, l'environnement d'exécution Lambda transmet deux arguments à la fonction, l'objet d'événement qui contient les données à traiter par votre fonction, et l'objet de contexte qui contient des informations sur l'invocation de la fonction.

Convention de nommage du gestionnaire

Le nom du gestionnaire de fonction défini au moment de la création d'une fonction Lambda est dérivé des éléments suivants :

- Nom du fichier dans lequel se trouve la fonction du gestionnaire Lambda.

- Nom de la fonction du gestionnaire Python.

Dans l'exemple ci-dessus, si le fichier est nommé `lambda_function.py`, le gestionnaire sera spécifié comme `lambda_function.lambda_handler`. Il s'agit du nom de gestionnaire par défaut donné aux fonctions que vous créez avec la console Lambda.

Si vous créez une fonction dans la console en utilisant un nom de fichier ou un nom de gestionnaire de fonction différent, vous devez modifier le nom du gestionnaire par défaut.

Pour modifier le nom du gestionnaire de fonction (console)

1. Ouvrez la page Fonctions de la console Lambda et choisissez votre fonction.
2. Cliquez sur l'onglet Code.
3. Faites défiler l'écran jusqu'au volet Paramètres d'exécution et choisissez Modifier.
4. Dans Gestionnaire, saisissez le nouveau nom de votre gestionnaire de fonction.
5. Choisissez Enregistrer.

Utilisation de l'objet d'événement Lambda

Lorsque Lambda invoque votre fonction, il transmet un argument d'objet d'événement au gestionnaire de fonction. Les objets JSON constituent le format d'événement le plus courant pour les fonctions Lambda. Dans l'exemple de code de la section précédente, la fonction exige une entrée au format suivant :

```
{
    "Order_id": "12345",
    "Amount": 199.99,
    "Item": "Wireless Headphones"
}
```

Si votre fonction est invoquée par un autre Service AWS, l'événement d'entrée est également un objet JSON. Le format exact de l'objet d'événement dépend du service qui invoque votre fonction. Pour connaître le format de l'événement pour un service en particulier, reportez-vous à la page appropriée du chapitre Intégration d'autres services.

Si l'événement d'entrée est sous la forme d'un objet JSON, le moteur d'exécution Lambda le convertit en dictionnaire Python. Pour attribuer des valeurs dans le JSON d'entrée aux variables de votre code, utilisez les méthodes standard des dictionnaires Python, comme illustré dans l'exemple de code.

Vous pouvez également transmettre des données à votre fonction sous forme de tableau JSON ou de n'importe quel autre type de données JSON valide. Le tableau suivant définit la manière dont le moteur d'exécution Python convertit ces types JSON.

Type de données JSON	Type de données Python
objet	dictionnaire (`dict`)
array	liste (`list`)
nombre	entier (`int`) ou nombre à virgule flottante (`float`)
chaîne	chaîne (`str`)
Booléen	Booléen (`bool`)
null	sans type (`NoneType`)

Accès et utilisation de l'objet de contexte Lambda

L'objet de contexte Lambda contient des informations sur la fonction, l'invocation et l'environnement d'exécution. Lambda transmet automatiquement l'objet de contexte à votre fonction lors de son invocation. Vous pouvez utiliser l'objet de contexte pour générer des informations sur l'invocation de votre fonction à des fins de surveillance.

L'objet de contexte est une classe Python définie dans le client d'interface d'environnement d'exécution Lambda. Pour renvoyer la valeur de l'une des propriétés de l'objet de contexte, utilisez la méthode correspondante sur l'objet de contexte. Par exemple, l'extrait de code suivant attribue la valeur de la propriété `aws_request_id` (l'identifiant de la demande d'invocation) à une variable nommée `request`.

```
request = context.aws_request_id
```

Pour en savoir plus sur l'utilisation de l'objet de contexte Lambda et pour consulter la liste complète des méthodes et propriétés disponibles, consultez the section called "Contexte".

Signatures de gestionnaire valides pour les gestionnaires Python

Lorsque vous définissez votre fonction de gestionnaire en Python, la fonction doit prendre deux arguments. Le premier de ces arguments est l'objet d'événement Lambda, et le second est l'objet de contexte Lambda. Par convention, ces arguments d'entrée sont généralement nommés `event` et `context`, mais vous pouvez leur donner le nom que vous souhaitez. Si vous déclarez votre fonction de gestionnaire avec un seul argument d'entrée, Lambda générera une erreur lorsqu'il tentera d'exécuter votre fonction. La méthode la plus courante pour déclarer une fonction de gestionnaire en Python est la suivante :

```
def lambda_handler(event, context):
```

Vous pouvez également utiliser des indications de type Python dans votre déclaration de fonction, comme illustré dans l'exemple suivant :

```
from typing import Dict, Any

def lambda_handler(event: Dict[str, Any], context: Any) -> Dict[str, Any]:
```

Pour utiliser un typage AWS spécifique pour les événements générés par d'autres Services AWS et pour l'objet de contexte, ajoutez le package `aws-lambda-typing` au package de déploiement de votre fonction. Vous pouvez installer cette bibliothèque dans votre environnement de développement en exécutant **`pip install aws-lambda-typing`**. L'exemple de code suivant vous montre comment utiliser des indications de type spécifiques à AWS. Dans cet exemple, l'événement attendu est un événement Amazon S3.

```
from aws_lambda_typing.events import S3Event
from aws_lambda_typing.context import Context
from typing import Dict, Any

def lambda_handler(event: S3Event, context: Context) -> Dict[str, Any]:
```

Vous ne pouvez pas utiliser le type de fonction `async` Python pour votre fonction de gestionnaire.

Renvoi d'une valeur

Un gestionnaire peut renvoyer une valeur, qui doit être sérialisable en JSON. Les types de retour les plus courants comprennent `dict`, `list`, `str`, `int`, `float` et `bool`.

Ce qu'il advient de la valeur renvoyée dépend du type d'invocation et du service qui a invoqué la fonction. Par exemple :

- Si vous utilisez le type d'invocation `RequestResponse` pour invoquer une fonction Lambda de façon asynchrone, Lambda renvoie le résultat de l'appel de la fonction Python au client qui invoque la fonction Lambda (dans la réponse HTTP à la demande d'invocation, sérialisée dans le JSON). Par exemple, la console AWS Lambda utilise le type d'invocation `RequestResponse`. Dès lors, lorsque vous invoquez la fonction par le biais de la console, cette dernière affiche la valeur renvoyée.
- Si le gestionnaire renvoie des objets qui ne peuvent pas être sérialisées par `json.dumps`, le runtime renvoie une erreur.
- Si le gestionnaire renvoie `None`, comme le font implicitement les fonctions Python sans une instruction `return`, le runtime renvoie `null`.
- Si vous exécutez le type d'invocation `Event` (une invocation asynchrone), la valeur sera ignorée.

Dans l'exemple de code, le gestionnaire renvoie le dictionnaire Python suivant :

```
{
  "statusCode": 200,
  "message": "Receipt processed successfully"
}
```

L'environnement d'exécution Lambda sérialise ce dictionnaire et le renvoie au client qui a invoqué la fonction sous forme de chaîne JSON.

> **Note**
>
> Dans Python 3.9 et les versions ultérieures, Lambda inclut le requestId de l'invocation dans la réponse d'erreur.

Utilisation d'AWS SDK pour Python (Boto3) dans votre gestionnaire

Vous utiliserez souvent les fonctions Lambda pour interagir avec d'autres ressources et Services AWS. Le moyen le plus simple d'interagir avec ces ressources est d'utiliser AWS SDK pour Python (Boto3). Tous les environnements d'exécution Python Lambda pris en charge incluent une version du SDK pour Python. Toutefois, nous vous recommandons vivement d'inclure le SDK dans le

package de déploiement de votre fonction si votre code doit l'utiliser. L'inclusion du SDK dans votre package de déploiement vous permet de contrôler totalement vos dépendances et de réduire le risque de problèmes de désalignement des versions avec les autres bibliothèques. Pour en savoir plus, consultez the section called "Dépendances d'exécution dans Python" et the section called "Rétrocompatibilité".

Pour utiliser le SDK pour Python dans votre fonction Lambda, ajoutez l'instruction suivante au bloc import au début du code de votre fonction :

```
import boto3
```

Utilisez la commande `pip install` pour ajouter la bibliothèque `boto3` au package de déploiement de votre fonction. Pour des instructions détaillées sur l'ajout de dépendances à un package de déploiement .zip, consultez the section called "Création d'un package de déploiement .zip avec dépendances". Pour en savoir plus sur l'ajout de dépendances aux fonctions Lambda déployées sous forme d'images de conteneur, consultez the section called "Création d'une image à partir d'une image de base" ou the section called "Création d'une image à partir d'une image de base alternative".

Lorsque vous utilisez `boto3` dans votre code, vous n'avez pas besoin de fournir des informations d'identification pour initialiser un client. Par exemple, dans l'exemple de code, nous utilisons la ligne de code suivante pour initialiser un client Amazon S3 :

```
# Initialize the S3 client outside of the handler
s3_client = boto3.client('s3')
```

Avec Python, Lambda crée automatiquement des variables d'environnement avec des informations d'identification. Le SDK `boto3` vérifie la présence de ces informations d'identification dans les variables d'environnement de votre fonction lors de l'initialisation.

Accès aux variables d'environnement

Dans le code de votre gestionnaire, vous pouvez référencer les variables d'environnement à l'aide de la méthode `os.environ.get`. Dans l'exemple de code, nous référençons la variable d'environnement `RECEIPT_BUCKET` définie à l'aide de la ligne de code suivante :

```
# Access environment variables
bucket_name = os.environ.get('RECEIPT_BUCKET')
```

N'oubliez pas d'inclure une instruction `import os` dans le bloc import au début de votre code.

Pratiques exemplaires en matière de code pour les fonctions Lambda Python

Respectez les directives de la liste suivante pour utiliser les pratiques exemplaires de codage lors de la création de vos fonctions Lambda :

- Séparez le gestionnaire Lambda de votre logique principale. Cela vous permet de créer une fonction testable plus unitaire. Par exemple, en Python, vous pouvez observer ce qui suit :

```
def lambda_handler(event, context):
    foo = event['foo']
    bar = event['bar']
    result = my_lambda_function(foo, bar)

def my_lambda_function(foo, bar):
    // MyLambdaFunction logic here
```

- Contrôlez les dépendances du package de déploiement de vos fonctions. L'environnement d'exécution AWS Lambda contient un certain nombre de bibliothèques. Pour les environnements d'exécution Node.js et Python, ceux-ci incluent les kits SDK AWS. Pour activer le dernier ensemble de mises à jour des fonctionnalités et de la sécurité, Lambda met régulièrement à jour ces bibliothèques. Ces mises à jour peuvent introduire de subtiles modifications dans le comportement de votre fonction Lambda. Pour disposer du contrôle total des dépendances que votre fonction utilise, empaquetez toutes vos dépendances avec votre package de déploiement.
- Réduisez la complexité de vos dépendances. Privilégiez les infrastructures plus simples qui se chargent rapidement au démarrage de l'environnement d'exécution.
- Réduisez la taille de votre package de déploiement selon ses besoins relatifs à l'environnement d'exécution. Cela contribue à réduire le temps nécessaire au téléchargement et à la décompression de votre package de déploiement avant l'invocation.

Tirez parti de la réutilisation de l'environnement d'exécution pour améliorer les performances de votre fonction. Initialisez les clients SDK et les connexions à la base de données en dehors du gestionnaire de fonctions et mettez en cache les actifs statiques localement dans le répertoire `/tmp`. Les invocations ultérieures traitées par la même instance de votre fonction peuvent réutiliser ces ressources. Cela permet d'économiser des coûts, tout en réduisant le temps d'exécution de la fonction.

Pour éviter des éventuelles fuites de données entre les invocations, n'utilisez pas l'environnement d'exécution pour stocker des données utilisateur, des événements ou d'autres informations ayant un impact sur la sécurité. Si votre fonction repose sur un état réversible qui ne peut pas être stocké en mémoire dans le gestionnaire, envisagez de créer une fonction distincte ou des versions distinctes d'une fonction pour chaque utilisateur.

Utilisez une directive keep-alive pour maintenir les connexions persistantes. Lambda purge les connexions inactives au fil du temps. Si vous tentez de réutiliser une connexion inactive lorsque vous invoquez une fonction, cela entraîne une erreur de connexion. Pour maintenir votre connexion persistante, utilisez la directive Keep-alive associée à votre environnement d'exécution. Pour obtenir un exemple, consultez Réutilisation des connexions avec Keep-Alive dans Node.js.

Utilisez des variables d'environnement pour transmettre des paramètres opérationnels à votre fonction. Par exemple, si vous écrivez dans un compartiment Amazon S3 au lieu de coder en dur le nom du compartiment dans lequel vous écrivez, configurez le nom du compartiment comme variable d'environnement.

Évitez d'utiliser des invocations récursives dans votre fonction Lambda, lorsque la fonction s'invoque elle-même ou démarre un processus susceptible de l'invoquer à nouveau. Cela peut entraîner un volume involontaire d'invocations de fonction et des coûts accrus. Si vous constatez un volume involontaire d'invocations, définissez immédiatement la simultanéité réservée à la fonction sur `0` afin de limiter toutes les invocations de la fonction, pendant que vous mettez à jour le code.

N'utilisez pas d'API non publiques non documentées dans le code de votre fonction Lambda. Pour les exécutions gérées AWS Lambda, Lambda applique périodiquement des mises à jour de sécurité et fonctionnelles aux API internes de Lambda. Ces mises à jour internes de l'API peuvent être incompatibles avec les versions antérieures, entraînant des conséquences imprévues telles que des échecs d'invocation si votre fonction dépend de ces API non publiques. Consultez la Référence d'API pour obtenir la liste des API accessibles au public.

Écriture du code idempotent. L'écriture de code idempotent pour vos fonctions garantit ne gestion identique des événements dupliqués. Votre code doit valider correctement les événements et gérer correctement les événements dupliqués. Pour de plus amples informations, veuillez consulterComment faire en sorte que ma fonction Lambda soit idempotente ?.

Travailler avec des archives de fichiers .zip pour les fonctions Lambda Python

Le code de votre AWS Lambda fonction comprend un fichier .py contenant le code du gestionnaire de votre fonction, ainsi que les packages et modules supplémentaires dont dépend votre code. Pour déployer ce code de fonction vers Lambda, vous utilisez un package de déploiement. Ce package peut être une archive de fichier .zip ou une image de conteneur. Pour plus d'informations sur l'utilisation d'images de conteneur avec Python, consultez la page Déployer des fonctions Lambda Python avec des images de conteneur.

Pour créer votre package de déploiement sous forme d'archive de fichier .zip, vous pouvez utiliser l'utilitaire d'archivage .zip intégré à votre outil de ligne de commande, ou tout autre utilitaire .zip tel que 7zip. Les exemples présentés dans les sections suivantes supposent que vous utilisez un outil `zip` de ligne de commande dans un environnement Linux ou macOS. Pour utiliser les mêmes commandes sous Windows, vous pouvez installer le sous-système Windows pour Linux afin d'obtenir une version intégrée à Windows d'Ubuntu et de Bash.

Notez que Lambda utilise les autorisations de fichiers POSIX. Ainsi, vous pourriez devoir définir des autorisations pour le dossier du package de déploiement avant de créer l'archive de fichiers .zip.

Rubriques

- Dépendances d'exécution dans Python
- Création d'un package de déploiement .zip sans dépendances
- Création d'un package de déploiement .zip avec dépendances
- Chemin de recherche des dépendances et bibliothèques incluses dans l'exécution
- Utilisation des dossiers __pycache__
- Création de packages de déploiement .zip avec des bibliothèques natives
- Création et mise à jour de fonctions Lambda Python à l'aide de fichiers .zip

Dépendances d'exécution dans Python

Pour les fonctions Lambda qui utilisent l'exécution Python, une dépendance peut être n'importe quel package ou module Python. Lorsque vous déployez votre fonction à l'aide d'une archive .zip, vous pouvez soit ajouter ces dépendances à votre fichier .zip avec votre code de fonction, soit utiliser une couche Lambda. Une couche est un fichier .zip séparé qui peut contenir du code supplémentaire et

d'autres contenus. Pour en savoir plus sur l'utilisation des couches Lambda dans Python, consultez the section called "Layers".

Les environnements d'exécution Lambda Python incluent le AWS SDK pour Python (Boto3) et ses dépendances. Lambda fournit le kit SDK dans l'environnement d'exécution pour les scénarios de déploiement où vous n'êtes pas en mesure d'ajouter vos propres dépendances. Ces scénarios incluent la création de fonctions dans la console à l'aide de l'éditeur de code intégré ou à l'utilisation de fonctions intégrées dans AWS Serverless Application Model (AWS SAM) ou de CloudFormation modèles.

Lambda met régulièrement à jour les bibliothèques d'exécution Python afin d'inclure les dernières mises à jour et correctifs de sécurité. Si votre fonction utilise la version du kit SDK Boto3 incluse dans l'exécution, mais que votre package de déploiement inclut des dépendances du kit SDK, cela peut entraîner des problèmes d'alignement de versions. Par exemple, votre package de déploiement peut inclure la dépendance du kit SDK urllib3. Lorsque Lambda met à jour le kit SDK dans l'exécution, des problèmes de compatibilité entre la nouvelle version de l'exécution et la version d'urllib3 de votre package de déploiement peuvent faire échouer votre fonction.

> **Important**
>
> Pour conserver un contrôle total sur vos dépendances et éviter d'éventuels problèmes d'alignement de versions, nous vous recommandons d'ajouter toutes les dépendances de votre fonction à votre package de déploiement, même si des versions de celles-ci sont incluses dans l'exécution Lambda. Cela inclut le kit SDK Boto3.

Pour savoir quelle version du kit SDK pour Python (Boto3) est incluse dans l'environnement d'exécution que vous utilisez, consultez the section called "Versions du SDK incluses dans l'environnement d'exécution".

Dans le cadre du modèle de responsabilité partagée AWS, vous êtes responsable de la gestion de toutes les dépendances dans les packages de déploiement de vos fonctions. Cela inclut l'application de mises à jour et de correctifs de sécurité. Pour mettre à jour les dépendances dans le package de déploiement de votre fonction, créez d'abord un nouveau fichier .zip, puis chargez-le sur Lambda. Pour plus d'informations, consultez Création d'un package de déploiement .zip avec dépendances et Création et mise à jour de fonctions Lambda Python à l'aide de fichiers .zip.

Création d'un package de déploiement .zip sans dépendances

Si le code de votre fonction ne comporte aucune dépendance, votre fichier .zip contient uniquement le fichier .py contenant le code du gestionnaire de votre fonction. Utilisez votre utilitaire zip préféré pour créer un fichier .zip avec votre fichier .py à la racine. Si le fichier .py ne se trouve pas à la racine de votre fichier .zip, Lambda ne sera pas en mesure d'exécuter votre code.

Pour savoir comment déployer votre fichier .zip pour créer une nouvelle fonction Lambda ou mettre à jour une fonction Lambda existante, veuillez consulter la rubrique Création et mise à jour de fonctions Lambda Python à l'aide de fichiers .zip.

Création d'un package de déploiement .zip avec dépendances

Si votre code de fonction dépend de packages ou de modules supplémentaires, vous pouvez soit ajouter ces dépendances à votre fichier .zip avec votre code de fonction, soit utiliser une couche Lambda. Les instructions de cette section vous indiquent comment inclure vos dépendances dans votre package de déploiement .zip. Pour que Lambda puisse exécuter votre code, le fichier .py contenant le code de votre gestionnaire et toutes les dépendances de votre fonction doit être installé à la racine du fichier .zip.

Supposons que votre code de fonction soit enregistré dans un fichier nommé `lambda_function.py`. L'exemple de commandes d'interface de ligne de commande suivant crée un fichier .zip nommé `my_deployment_package.zip` contenant le code de votre fonction et ses dépendances. Vous pouvez soit installer vos dépendances directement dans un dossier du répertoire de votre projet, soit utiliser un environnement virtuel Python.

Pour créer le package de déploiement (répertoire du projet)

1. Accédez au répertoire du projet qui contient votre fichier de code source `lambda_function.py`. Dans cet exemple, le répertoire est nommé `my_function`.

   ```
   cd my_function
   ```

2. Créez un nouveau répertoire nommé « package » dans lequel vous installerez vos dépendances.

   ```
   mkdir package
   ```

Notez que pour un package de déploiement .zip, Lambda s'attend à ce que votre code source et ses dépendances se trouvent tous à la racine du fichier .zip. Toutefois, l'installation de dépendances directement dans le répertoire de votre projet peut introduire un grand nombre de nouveaux fichiers et dossiers et rendre la navigation dans votre IDE difficile. Vous créez ici un répertoire `package` distinct pour séparer vos dépendances de votre code source.

3. Installez les dépendances dans le répertoire `package`. L'exemple ci-dessous installe le kit SDK Boto3 à partir de l'index du Python Package Index à l'aide de pip. Si le code de votre fonction utilise des packages Python que vous avez créés vous-même, enregistrez-les dans le répertoire `package`.

```
pip install --target ./package boto3
```

4. Créez un fichier .zip avec les bibliothèques installées à la racine.

```
cd package
zip -r ../my_deployment_package.zip .
```

Cela génère un fichier `my_deployment_package.zip` dans votre répertoire de projet.

5. Ajoutez le fichier lambda_function.py à la racine du fichier .zip.

```
cd ..
zip my_deployment_package.zip lambda_function.py
```

Votre fichier .zip doit avoir une structure de répertoires horizontale, avec le code du gestionnaire de votre fonction et tous vos dossiers de dépendances installés à la racine comme suit.

```
my_deployment_package.zip
|- bin
|  |-jp.py
|- boto3
|  |-compat.py
|  |-data
|  |-docs
...
|- lambda_function.py
```

Si le fichier .py contenant le code du gestionnaire de votre fonction ne se trouve pas à la racine de votre fichier .zip, Lambda ne sera pas en mesure d'exécuter votre code.

Pour créer le package de déploiement (environnement virtuel)

1. Créez et activez un environnement virtuel dans le répertoire de votre projet. Dans cet exemple, le répertoire du projet est nommé `my_function`.

   ```
   ~$ cd my_function
   ~/my_function$ python3.14 -m venv my_virtual_env
   ~/my_function$ source ./my_virtual_env/bin/activate
   ```

2. Installez les bibliothèques requises à l'aide de pip. L'exemple suivant permet d'installer le kit SDK Boto3

   ```
   (my_virtual_env) ~/my_function$ pip install boto3
   ```

3. Utilisez `pip show` pour trouver l'emplacement dans votre environnement virtuel où pip a installé vos dépendances.

   ```
   (my_virtual_env) ~/my_function$ pip show <package_name>
   ```

   Le dossier dans lequel pip installe vos bibliothèques peut être nommé `site-packages` ou `dist-packages`. Ce dossier peut se trouver dans le répertoire `lib/python3.x` ou `lib64/python3.x` (où python3.x représente la version de Python que vous utilisez).

4. Désactivation de l'environnement virtuel

   ```
   (my_virtual_env) ~/my_function$ deactivate
   ```

5. Accédez au répertoire contenant les dépendances que vous avez installées avec pip et créez un fichier .zip dans le répertoire de votre projet avec les dépendances installées à la racine. Dans cet exemple, pip a installé vos dépendances dans le répertoire `my_virtual_env/lib/python3.14/site-packages`.

   ```
   ~/my_function$ cd my_virtual_env/lib/python3.14/site-packages
   ~/my_function/my_virtual_env/lib/python3.14/site-packages$ zip -r ../../../../
   my_deployment_package.zip .
   ```

6. Accédez à la racine du répertoire de votre projet où se trouve le fichier .py contenant le code de votre gestionnaire et ajoutez ce fichier à la racine de votre package .zip. Dans cet exemple, votre fichier de code de fonction est nommé `lambda_function.py`.

```
~/my_function/my_virtual_env/lib/python3.14/site-packages$ cd ../../../../
~/my_function$ zip my_deployment_package.zip lambda_function.py
```

Chemin de recherche des dépendances et bibliothèques incluses dans l'exécution

Lorsque vous utilisez une instruction `import` dans votre code, l'exécution Python recherche les répertoires dans son chemin de recherche jusqu'à ce qu'elle trouve le module ou le package. Par défaut, le premier emplacement dans lequel recherche l'exécution est le répertoire dans lequel votre package de déploiement .zip est décompressé et monté (`/var/task`). Si vous ajoutez une version d'une bibliothèque incluse dans l'exécution dans votre package de déploiement, votre version aura la priorité sur la version incluse dans l'exécution. Les dépendances de votre package de déploiement ont également la priorité sur les dépendances des couches.

Lorsque vous ajoutez une dépendance à une couche, Lambda l'extrait vers `/opt/python/lib/python3.x/site-packages` (où `python3.x` représente la version de l'exécution que vous utilisez) ou `/opt/python`. Dans le chemin de recherche, ces répertoires ont la priorité sur les répertoires contenant les bibliothèques incluses dans l'exécution et les bibliothèques installées par pip (`/var/runtime` et `/var/lang/lib/python3.x/site-packages`). Les bibliothèques des couches de fonctions ont donc la priorité sur les versions incluses dans l'exécution.

> **Note**
>
> Dans le runtime géré et l'image de base de Python 3.11, le AWS SDK et ses dépendances sont installés dans le `/var/lang/lib/python3.11/site-packages` répertoire.

Vous pouvez voir le chemin de recherche complet de votre fonction Lambda en ajoutant l'extrait de code suivant.

```
import sys

search_path = sys.path
print(search_path)
```

> **Note**
>
> Étant donné que les dépendances de votre package de déploiement ou de vos couches sont prioritaires sur les bibliothèques incluses dans l'exécution, cela peut entraîner des problèmes d'alignement de versions si vous incluez une dépendance du kit SDK telle que urllib3 dans votre package sans inclure également le kit SDK. Si vous déployez votre propre version d'une dépendance de Boto3, vous devez également déployer Boto3 en tant que dépendance dans votre package de déploiement. Nous vous recommandons de regrouper toutes les dépendances de votre fonction, même si des versions de celles-ci sont incluses dans l'exécution.

Vous pouvez également ajouter des dépendances dans un dossier distinct au sein de votre package .zip. Par exemple, vous pouvez ajouter une version du kit SDK Boto3 dans un dossier de votre package .zip appelé `common`. Lorsque votre package .zip est décompressé et monté, ce dossier est placé dans le répertoire `/var/task`. Pour utiliser dans votre code une dépendance provenant d'un dossier de votre package de déploiement .zip, utilisez une instruction `import from`. Par exemple, pour utiliser une version de Boto3 provenant d'un dossier nommé `common` dans votre package .zip, utilisez l'instruction suivante.

```
from common import boto3
```

Utilisation des dossiers __pycache__

Nous vous recommandons de ne pas inclure de dossiers `__pycache__` dans le package de déploiement de votre fonction. Le bytecode Python compilé sur une machine de génération avec une architecture ou un système d'exploitation différent peut ne pas être compatible avec l'environnement d'exécution Lambda.

Création de packages de déploiement .zip avec des bibliothèques natives

Si votre fonction utilise uniquement des packages et des modules Python purs, vous pouvez utiliser la commande `pip install` pour installer vos dépendances sur n'importe quelle machine de génération locale et créer votre fichier .zip. De nombreuses bibliothèques Python populaires, y compris NumPy Pandas, ne sont pas du Python pur et contiennent du code écrit en C ou C++. Lorsque vous ajoutez des bibliothèques contenant du code C/C++ à votre package de déploiement, vous devez créer votre package correctement pour vous assurer qu'il est compatible avec l'environnement d'exécution Lambda.

La plupart des packages disponibles sur le Python Package Index (PyPI) sont disponibles sous forme de « wheels » (fichiers .whl). Un fichier .whl est un type de fichier ZIP qui contient une distribution intégrée avec des fichiers binaires précompilés pour un système d'exploitation et une architecture de l'ensemble des instructions particuliers. Pour rendre votre package de déploiement compatible avec Lambda, vous devez installer le fichier wheel pour les systèmes d'exploitation Linux et l'architecture de l'ensemble d'instructions de votre fonction.

Certains packages peuvent uniquement être disponibles en tant que distributions sources. Pour ces packages, vous devez compiler et créer vous-même les composants C/C++.

Pour connaître les distributions disponibles pour le package dont vous avez besoin, procédez comme suit :

1. Recherchez le nom du package sur la page principale du Python Package Index.
2. Choisissez la version du package que vous souhaitez utiliser.
3. Choisissez Télécharger les fichiers.

Utilisation des distributions intégrées (fichiers wheels)

Pour télécharger un fichier wheel compatible avec Lambda, utilisez l'option `--platform` pip.

Si votre fonction Lambda utilise l'architecture de l'ensemble d'instructions x86_64, exécutez la commande `pip install` suivante pour installer un fichier wheel compatible dans votre répertoire `package`. Remplacez `--python 3.x` par la version d'exécution Python que vous utilisez.

```
pip install \
--platform manylinux2014_x86_64 \
--target=package \
--implementation cp \
--python-version 3.x \
--only-binary=:all: --upgrade \
<package_name>
```

Si votre fonction utilise l'architecture de l'ensemble d'instructions arm64, exécutez la commande suivante. Remplacez `--python 3.x` par la version d'exécution Python que vous utilisez.

```
pip install \
--platform manylinux2014_aarch64 \
--target=package \
```

```
--implementation cp \
--python-version 3.x \
--only-binary=:all: --upgrade \
<package_name>
```

Utilisation des distributions sources

Si votre package n'est disponible que sous la forme d'une distribution source, vous devez créer vous-même les bibliothèques C/C++. Pour rendre votre package compatible avec l'environnement d'exécution Lambda, vous devez le créer dans un environnement qui utilise le même système d'exploitation Amazon Linux. Pour ce faire, créez votre package dans une instance Linux Amazon Elastic Compute Cloud (Amazon EC2).

Pour savoir comment lancer une instance Amazon EC2 Linux et s'y connecter, consultez la section Commencer avec Amazon EC2 dans le guide de EC2 l'utilisateur Amazon.

Création et mise à jour de fonctions Lambda Python à l'aide de fichiers .zip

Une fois que vous avez créé votre package de déploiement .zip, vous pouvez l'utiliser pour créer une nouvelle fonction Lambda ou mettre à jour une fonction Lambda existante. Vous pouvez déployer votre package .zip à l'aide de la console Lambda, de l'API Lambda et AWS Command Line Interface de l'API Lambda. Vous pouvez également créer et mettre à jour des fonctions Lambda à l'aide de l' AWS Serverless Application Model (AWS SAM) et de CloudFormation.

La taille maximale d'un package de déploiement .zip pour Lambda est de 250 Mo (décompressé). Notez que cette limite s'applique à la taille combinée de tous les fichiers que vous chargez, y compris les couches Lambda.

Le runtime Lambda a besoin d'une autorisation pour lire les fichiers de votre package de déploiement. Dans la notation octale des autorisations Linux, Lambda a besoin de 644 autorisations pour les fichiers non exécutables (rw-r--r--) et de 755 autorisations () pour les répertoires et les fichiers exécutables. rwxr-xr-x

Sous Linux et macOS, utilisez la commande `chmod` pour modifier les autorisations de fichiers sur les fichiers et les répertoires de votre package de déploiement. Par exemple, pour octroyer à un fichier non exécutable les autorisations correctes, exécutez la commande suivante.

```
chmod 644 <filepath>
```

Pour modifier les autorisations relatives aux fichiers dans Windows, voir Définir, afficher, modifier ou supprimer des autorisations sur un objet dans la documentation Microsoft Windows.

> **Note**
>
> Si vous n'accordez pas à Lambda les autorisations nécessaires pour accéder aux répertoires de votre package de déploiement, Lambda définit les autorisations pour ces répertoires sur 755 (). rwxr-xr-x

Création et mise à jour de fonctions avec des fichiers .zip à l'aide de la console

Pour créer une nouvelle fonction, vous devez d'abord créer la fonction dans la console, puis charger votre archive .zip. Pour mettre à jour une fonction existante, ouvrez la page de votre fonction, puis suivez la même procédure pour ajouter votre fichier .zip mis à jour.

Si votre fichier .zip fait moins de 50 Mo, vous pouvez créer ou mettre à jour une fonction en chargeant le fichier directement à partir de votre ordinateur local. Pour les fichiers .zip de plus de 50 Mo, vous devez d'abord charger votre package dans un compartiment Amazon S3. Pour savoir comment charger un fichier dans un compartiment Amazon S3 à l'aide du AWS Management Console, consultez Getting started with Amazon S3. Pour télécharger des fichiers à l'aide de AWS CLI, voir Déplacer des objets dans le guide de AWS CLI l'utilisateur.

> **Note**
>
> Vous ne pouvez pas modifier le type de package de déploiement (.zip ou image de conteneur) d'une fonction existante. Par exemple, vous ne pouvez pas convertir une fonction d'image de conteneur pour utiliser un fichier d'archive .zip à la place. Vous devez créer une nouvelle fonction.

Pour créer une nouvelle fonction (console)

1. Ouvrez la page Fonctions de la console Lambda et choisissez Créer une fonction.
2. Choisissez Créer à partir de zéro.
3. Sous Informations de base, procédez comme suit :

    a. Pour Nom de la fonction, saisissez le nom de la fonction.

b. Pour Exécution, sélectionnez l'exécution que vous souhaitez utiliser.

c. (Facultatif) Pour Architecture, choisissez l'architecture de l'ensemble des instructions pour votre fonction. L'architecture par défaut est x86_64. Assurez-vous que le package de déploiement .zip pour votre fonction est compatible avec l'architecture de l'ensemble d'instructions que vous sélectionnez.

4. (Facultatif) Sous Permissions (Autorisations), développez Change default execution role (Modifier le rôle d'exécution par défaut). Vous pouvez créer un rôle d'exécution ou en utiliser un existant.
5. Choisissez Créer une fonction. Lambda crée une fonction de base « Hello world » à l'aide de l'exécution de votre choix.

Pour charger une archive .zip à partir de votre ordinateur local (console)

1. Sur la page Fonctions de la console Lambda, choisissez la fonction pour laquelle vous souhaitez charger le fichier .zip.
2. Sélectionnez l'onglet Code.
3. Dans le volet Source du code, choisissez Charger à partir de.
4. Choisissez Fichier .zip.
5. Pour charger un fichier .zip, procédez comme suit :

   a. Sélectionnez Charger, puis choisissez votre fichier .zip dans le sélecteur de fichiers.

   b. Choisissez Ouvrir.

   c. Choisissez Enregistrer.

Pour charger une archive .zip depuis un compartiment Amazon S3 (console)

1. Sur la page Fonctions de la console Lambda, choisissez la fonction pour laquelle vous souhaitez charger un nouveau fichier .zip.
2. Sélectionnez l'onglet Code.
3. Dans le volet Source du code, choisissez Charger à partir de.
4. Choisissez l'emplacement Amazon S3.
5. Collez l'URL du lien Amazon S3 de votre fichier .zip et choisissez Enregistrer.

Mise à jour des fonctions du fichier .zip à l'aide de l'éditeur de code de la console

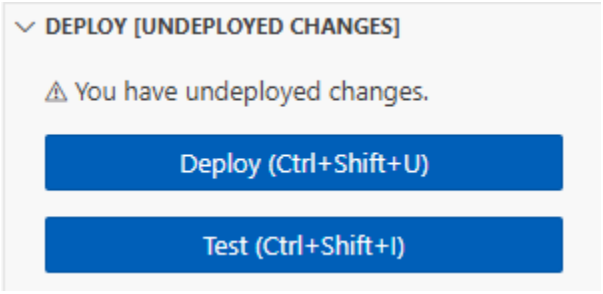
Pour certaines fonctions avec des packages de déploiement .zip, vous pouvez utiliser l'éditeur de code intégré de la console Lambda pour mettre à jour le code de votre fonction directement. Pour utiliser cette fonctionnalité, votre fonction doit répondre aux critères suivants :

- Votre fonction doit utiliser l'une des exécutions des langages interprétés (Python, Node.js ou Ruby).
- La taille du package de déploiement de votre fonction doit être inférieure à 50 Mo (décompressé).

Le code des fonctions avec les packages de déploiement d'images de conteneurs ne peut pas être édité directement dans la console.

Pour mettre à jour le code de fonction à l'aide de l'éditeur de code de la console

1. Ouvrez la page Fonctions de la console Lambda et choisissez votre fonction.
2. Sélectionnez l'onglet Code.
3. Dans le volet Source du code, sélectionnez votre fichier de code source et modifiez-le dans l'éditeur de code intégré.
4. Dans la section DÉPLOYER, choisissez Déployer pour mettre à jour le code de votre fonction :



Création et mise à jour de fonctions avec des fichiers .zip à l'aide du AWS CLI

Vous pouvez utiliser l'AWS CLI pour créer une nouvelle fonction ou pour mettre à jour une fonction existante à l'aide d'un fichier .zip. Utilisez la fonction de création et update-function-codeles commandes pour déployer votre package .zip. Si votre fichier .zip est inférieur à 50 Mo, vous pouvez charger le package .zip à partir d'un emplacement de fichier sur votre machine de génération locale. Pour les fichiers plus volumineux, vous devez charger votre package .zip à partir d'un compartiment Amazon S3. Pour savoir comment charger un fichier dans un compartiment Amazon S3 à l'aide du AWS CLI, consultez la section Déplacer des objets dans le guide de AWS CLI l'utilisateur.

> **Note**
>
> Si vous chargez votre fichier .zip depuis un compartiment Amazon S3 à l'aide de AWS CLI, le compartiment doit se trouver au même endroit Région AWS que votre fonction.

Pour créer une nouvelle fonction à l'aide d'un fichier .zip avec le AWS CLI, vous devez spécifier les éléments suivants :

- Le nom de votre fonction (`--function-name`)
- L'exécution de votre fonction (`--runtime`)
- L'Amazon Resource Name (ARN) du rôle d'exécution de votre fonction (`--role`)
- Le nom de la méthode du gestionnaire dans votre code de fonction (`--handler`)

Vous devez également indiquer l'emplacement de votre fichier .zip. Si votre fichier .zip se trouve dans un dossier sur votre machine de génération locale, utilisez l'option `--zip-file` pour spécifier le chemin d'accès du fichier, comme le montre l'exemple de commande suivant.

```
aws lambda create-function --function-name myFunction \
--runtime python3.14 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

Pour spécifier l'emplacement du fichier .zip dans un compartiment Amazon S3, utilisez l'option `--code` comme le montre l'exemple de commande suivant. Vous devez uniquement utiliser le paramètre `S3ObjectVersion` pour les objets soumis à la gestion des versions.

```
aws lambda create-function --function-name myFunction \
--runtime python3.14 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

Pour mettre à jour une fonction existante à l'aide de l'interface de ligne de commande, vous devez spécifier le nom de votre fonction à l'aide du paramètre `--function-name`. Vous devez également spécifier l'emplacement du fichier .zip que vous souhaitez utiliser pour mettre à jour votre code de fonction. Si votre fichier .zip se trouve dans un dossier sur votre machine de génération locale,

utilisez l'option `--zip-file` pour spécifier le chemin d'accès du fichier, comme le montre l'exemple de commande suivant.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Pour spécifier l'emplacement du fichier .zip dans un compartiment Amazon S3, utilisez les options `--s3-bucket` et `--s3-key` comme le montre l'exemple de commande suivant. Vous devez uniquement utiliser le paramètre `--s3-object-version` pour les objets soumis à la gestion des versions.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Création et mise à jour de fonctions avec des fichiers .zip à l'aide de l'API Lambda

Pour créer et mettre à jour des fonctions à l'aide d'une archive de fichiers .zip, utilisez les opérations d'API suivantes :

- CreateFunction
- UpdateFunctionCode

Création et mise à jour de fonctions avec des fichiers .zip à l'aide de AWS SAM

The AWS Serverless Application Model (AWS SAM) est une boîte à outils qui permet de rationaliser le processus de création et d'exécution d'applications sans serveur sur AWS. Vous définissez les ressources de votre application dans un modèle YAML ou JSON et vous utilisez l'interface de ligne de AWS SAM commande (AWS SAM CLI) pour créer, empaqueter et déployer vos applications. Lorsque vous créez une fonction Lambda à partir d'un AWS SAM modèle, elle crée AWS SAM automatiquement un package de déploiement ou une image de conteneur .zip avec le code de votre fonction et les dépendances que vous spécifiez. Pour en savoir plus sur l'utilisation des fonctions Lambda AWS SAM pour créer et déployer des fonctions Lambda, consultez Getting started with AWS SAM dans le Guide du AWS Serverless Application Model développeur.

Vous pouvez également créer une fonction Lambda AWS SAM à l'aide d'une archive de fichiers .zip existante. Pour créer une fonction Lambda à l'aide de AWS SAM, vous pouvez enregistrer votre fichier .zip dans un compartiment Amazon S3 ou dans un dossier local sur votre machine de

génération. Pour savoir comment charger un fichier dans un compartiment Amazon S3 à l'aide du AWS CLI, consultez la section Déplacer des objets dans le guide de AWS CLI l'utilisateur.

Dans votre AWS SAM modèle, la `AWS::Serverless::Function` ressource spécifie votre fonction Lambda. Dans cette ressource, définissez les propriétés suivantes pour créer une fonction à l'aide d'une archive de fichiers .zip :

- `PackageType` : défini sur `Zip`
- `CodeUri`- défini sur l'URI Amazon S3, le chemin d'accès au dossier local ou à l'FunctionCodeobjet du code de fonction
- `Runtime` : défini sur votre exécution choisie

Ainsi AWS SAM, si votre fichier .zip est supérieur à 50 Mo, vous n'avez pas besoin de le télécharger au préalable dans un compartiment Amazon S3. AWS SAM peut télécharger des packages .zip jusqu'à la taille maximale autorisée de 250 Mo (décompressés) à partir d'un emplacement sur votre machine de compilation locale.

Pour en savoir plus sur le déploiement de fonctions à l'aide d'un fichier .zip dans AWS SAM, consultez AWS::Serverless::Functionle manuel du AWS SAM développeur.

Création et mise à jour de fonctions avec des fichiers .zip à l'aide de CloudFormation

Vous pouvez l'utiliser CloudFormation pour créer une fonction Lambda à l'aide d'une archive de fichiers .zip. Pour créer une fonction Lambda à partir d'un fichier .zip, vous devez d'abord charger votre fichier dans un compartiment Amazon S3. Pour savoir comment charger un fichier dans un compartiment Amazon S3 à l'aide du AWS CLI, consultez la section Déplacer des objets dans le guide de AWS CLI l'utilisateur.

Pour les environnements d'exécution Node.js et Python, vous pouvez également fournir du code source intégré dans votre CloudFormation modèle. CloudFormation crée ensuite un fichier .zip contenant votre code lorsque vous créez votre fonction.

Utilisation d'un fichier .zip existant

Dans votre CloudFormation modèle, la `AWS::Lambda::Function` ressource spécifie votre fonction Lambda. Dans cette ressource, définissez les propriétés suivantes pour créer une fonction à l'aide d'une archive de fichiers .zip :

- `PackageType` : défini sur `Zip`

- `Code` : saisissez le nom du compartiment Amazon S3 et le nom du fichier .zip dans les champs `S3Bucket` et `S3Key`
- `Runtime` : défini sur votre exécution choisie

Création d'un fichier .zip à partir du code en ligne

Vous pouvez déclarer des fonctions simples écrites en Python ou en Node.js en ligne dans un CloudFormation modèle. Le code étant intégré dans YAML ou JSON, vous ne pouvez pas ajouter de dépendances externes à votre package de déploiement. Cela signifie que votre fonction doit utiliser la version du AWS SDK incluse dans le runtime. Les exigences du modèle, telles que l'obligation d'échapper certains caractères, rendent également plus difficile l'utilisation des fonctionnalités de vérification syntaxique et de complétion de code de votre IDE. Cela signifie que votre modèle peut nécessiter des tests supplémentaires. En raison de ces limitations, la déclaration de fonctions en ligne convient mieux à un code très simple qui ne change pas fréquemment.

Pour créer un fichier .zip à partir du code en ligne pour les exécutions Node.js et Python, définissez les propriétés suivantes dans la ressource `AWS::Lambda::Function` de votre modèle :

- `PackageType` : défini sur `Zip`
- `Code` : saisissez votre code de fonction dans le champ `ZipFile`
- `Runtime` : défini sur votre exécution choisie

Le fichier .zip CloudFormation généré ne peut pas dépasser 4 Mo. Pour en savoir plus sur le déploiement de fonctions à l'aide d'un fichier .zip dans CloudFormation, consultez AWS::Lambda::Functionle Guide de l'CloudFormation utilisateur.

Déployer des fonctions Lambda en Python avec des images conteneurs

Il existe trois méthodes pour créer une image de conteneur pour une fonction Lambda Python :

- Utilisation d'une image AWS de base pour Python

  Les images de base AWS sont préchargées avec une exécution du langage, un client d'interface d'exécution pour gérer l'interaction entre Lambda et votre code de fonction, et un émulateur d'interface d'exécution pour les tests locaux.

- Utilisation d'une image de base AWS uniquement pour le système d'exploitation

  AWS Les images de base réservées au système d'exploitation contiennent une distribution Amazon Linux et l'émulateur d'interface d'exécution. Ces images sont couramment utilisées pour créer des images de conteneur pour les langages compilés, tels que Go et Rust, et pour une langue ou une version linguistique pour laquelle Lambda ne fournit pas d'image de base, comme Node.js 19. Vous pouvez également utiliser des images de base uniquement pour le système d'exploitation pour implémenter un environnement d'exécution personnalisé. Pour rendre l'image compatible avec Lambda, vous devez inclure le client d'interface d'exécution pour Python dans l'image.

- Utilisation d'une image non AWS basique

  Vous pouvez utiliser une autre image de base à partir d'un autre registre de conteneur, comme Alpine Linux ou Debian. Vous pouvez également utiliser une image personnalisée créée par votre organisation. Pour rendre l'image compatible avec Lambda, vous devez inclure le client d'interface d'exécution pour Python dans l'image.

> **Tip**
>
> Pour réduire le temps nécessaire à l'activation des fonctions du conteneur Lambda, consultez Utiliser des générations en plusieurs étapes (français non garanti) dans la documentation Docker. Pour créer des images de conteneur efficaces, suivez la section Bonnes pratiques pour l'écriture de Dockerfiles (français non garanti).

Cette page explique comment créer, tester et déployer des images de conteneur pour Lambda.

Rubriques

- AWS images de base pour Python
- Utilisation d'une image AWS de base pour Python
- Utilisation d’une autre image de base avec le client d’interface d’exécution

AWS images de base pour Python

AWS fournit les images de base suivantes pour Python :

Étiquettes	Environnement d’exécution	Système d’exploitation	Dockerfile	Obsolescence
3,14	Python 3.14	Amazon Linux	Dockerfile pour Python 3.14 sur GitHub	30 juin 2029
3.13	Python 3.13	Amazon Linux	Dockerfile pour Python 3.13 sur GitHub	30 juin 2029
3,12	Python 3.12	Amazon Linux	Dockerfile pour Python 3.12 sur GitHub	31 octobre 2028
3,11	Python 3.11	Amazon Linux 2	Dockerfile pour Python 3.11 sur GitHub	30 juin 2027
3,10	Python 3.10	Amazon Linux 2	Dockerfile pour Python 3.10 sur GitHub	31 octobre 2026

Référentiel Amazon ECR : gallery.ecr. aws/lambda/python

Les images de base de Python 3.12 et versions ultérieures sont basées sur l’image de conteneur minimale Amazon Linux 2023. Les images de base Python 3.8-3.11 sont basées sur l'image Amazon Linux 2. AL2Les images basées sur la version 023 présentent plusieurs avantages par rapport à Amazon Linux 2, notamment un encombrement de déploiement réduit et des versions mises à jour de bibliothèques telles que. `glibc`

AL2Les images basées sur le format 023 sont utilisées `microdnf` (en lien symbolique comme`dnf`) comme gestionnaire de packages au lieu de`yum`, qui est le gestionnaire de packages par défaut dans Amazon Linux 2. `microdnf`est une implémentation autonome de. `dnf` Pour obtenir la liste des packages inclus dans les images AL2 basées sur la version 023, reportez-vous aux colonnes Conteneur minimal de la section Comparaison des packages installés sur les images de conteneurs Amazon Linux 2023. Pour plus d'informations sur les différences entre AL2 023 et Amazon Linux 2, consultez Présentation du runtime Amazon Linux 2023 AWS Lambda sur le blog AWS Compute.

> **Note**
>
> Pour exécuter des images AL2 basées sur 023 localement, y compris avec AWS Serverless Application Model (AWS SAM), vous devez utiliser Docker version 20.10.10 ou ultérieure.

Chemin de recherche des dépendances dans les images de base

Lorsque vous utilisez une instruction `import` dans votre code, l'exécution Python recherche les répertoires dans son chemin de recherche jusqu'à ce qu'elle trouve le module ou le package. Par défaut, l'exécution recherche d'abord dans le répertoire `{LAMBDA_TASK_ROOT}`. Si vous ajoutez une version d'une bibliothèque incluse dans l'exécution dans votre image, votre version aura la priorité sur la version incluse dans l'exécution.

Les autres étapes du chemin de recherche dépendent de la version de l'image de base Lambda pour Python que vous utilisez :

- Python 3.11 et versions ultérieures : Les bibliothèques incluses dans l'exécution et les bibliothèques installées par pip sont installées dans le répertoire `/var/lang/lib/python3.11/site-packages`. Ce répertoire a la priorité sur `/var/runtime` dans le chemin de recherche. Vous pouvez remplacer le kit SDK en utilisant pip pour installer une version plus récente. Vous pouvez utiliser pip pour vérifier que le kit SDK inclus dans l''exécution et ses dépendances sont compatibles avec tous les packages que vous installez.
- Python 3.8-3.10 : Les bibliothèques incluses dans l'exécution sont installées dans le répertoire `/var/runtime`. Les bibliothèques installées par PIP sont installées dans le répertoire `/var/lang/lib/python3.x/site-packages`. Le répertoire `/var/runtime` a la priorité sur `/var/lang/lib/python3.x/site-packages` dans le chemin de recherche.

Vous pouvez voir le chemin de recherche complet de votre fonction Lambda en ajoutant l'extrait de code suivant.

```
import sys

search_path = sys.path
print(search_path)
```

Utilisation d'une image AWS de base pour Python

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- AWS CLI version 2
- Docker (version minimale 25.0.0)
- Le plug-in buildx Docker.
- Python

Création d’une image à partir d’une image de base

Pour créer une image de conteneur à partir d'une image AWS de base pour Python

1. Créez un répertoire pour le projet, puis passez à ce répertoire.

   ```
   mkdir example
   cd example
   ```

2. Créez un nouveau fichier appelé `lambda_function.py`. Vous pouvez ajouter l’exemple de code de fonction suivant au fichier pour le tester, ou utiliser le vôtre.

   Example Fonction Python

   ```
   import sys
   def handler(event, context):
       return 'Hello from AWS Lambda using Python' + sys.version + '!'
   ```

3. Créez un nouveau fichier appelé `requirements.txt`. Si vous utilisez l’exemple de code de fonction de l’étape précédente, vous pouvez laisser le fichier vide, car il n’y a pas de dépendances. Sinon, répertoriez chaque bibliothèque requise. Par exemple, voici à quoi votre

fichier `requirements.txt` doit ressembler si votre fonction utilise AWS SDK pour Python (Boto3) :

Example requirements.txt

```
boto3
```

4. Créez un nouveau Dockerfile avec la configuration suivante :

   - Définir la propriété `FROM` sur l'URI de l'image de base.
   - Utilisez la commande COPY pour copier le code de la fonction et les dépendances de l'environnement d'exécution dans `{LAMBDA_TASK_ROOT}`, une variable d'environnement définie par Lambda.
   - Définir l'argument `CMD` pour le gestionnaire de la fonction Lambda.

   Notez que l'exemple de Dockerfile n'inclut pas d'instruction USER. Lorsque vous déployez une image de conteneur sur Lambda, Lambda définit automatiquement un utilisateur Linux par défaut disposant d'autorisations de moindre privilège. Ceci est différent du comportement standard de Docker qui est défini par défaut par l'utilisateur `root` lorsqu'aucune instruction USER n'est fournie.

   Example Dockerfile

```
FROM public.ecr.aws/lambda/python:3.12

# Copy requirements.txt
COPY requirements.txt ${LAMBDA_TASK_ROOT}

# Install the specified packages
RUN pip install -r requirements.txt

# Copy function code
COPY lambda_function.py ${LAMBDA_TASK_ROOT}

# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "lambda_function.handler" ]
```

5. Générez l'image Docker à l'aide de la commande docker build. L'exemple suivant nomme l'image `docker-image` et lui donne la balise `test`. Pour rendre votre image compatible avec Lambda, vous devez utiliser l'option `--provenance=false`.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> La commande spécifie l'option `--platform linux/amd64` pour garantir la compatibilité de votre conteneur avec l'environnement d'exécution Lambda, quelle que soit l'architecture de votre machine de génération. Si vous avez l'intention de créer une fonction Lambda à l'aide de l'architecture du jeu ARM64 d'instructions, veillez à modifier la commande pour utiliser l'`--platform linux/arm64`option à la place.

(Facultatif) Testez l'image localement

1. Démarrez votre image Docker à l'aide de la commande docker run. Dans cet exemple, `docker-image` est le nom de l'image et `test` est la balise.

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

Cette commande exécute l'image en tant que conteneur et crée un point de terminaison local à `localhost:9000/2015-03-31/functions/function/invocations`.

> **Note**
>
> Si vous avez créé l'image Docker pour l'architecture du jeu ARM64 d'instructions, veillez à utiliser l'`--platform linux/arm64`option au lieu de`--platform linux/amd64`.

2. À partir d'une nouvelle fenêtre de terminal, publiez un événement au point de terminaison local.

Linux/macOS

Sous Linux et macOS, exécutez la commande `curl` suivante :

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

Cette commande invoque la fonction avec un événement vide et renvoie une réponse. Si vous utilisez votre propre code de fonction plutôt que l'exemple de code de fonction, vous pouvez invoquer la fonction avec une charge utile JSON. Exemple :

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

Dans PowerShell, exécutez la `Invoke-WebRequest` commande suivante :

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Cette commande invoque la fonction avec un événement vide et renvoie une réponse. Si vous utilisez votre propre code de fonction plutôt que l'exemple de code de fonction, vous pouvez invoquer la fonction avec une charge utile JSON. Exemple :

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. Obtenez l'ID du conteneur.

```
docker ps
```

4. Utilisez la commande docker kill pour arrêter le conteneur. Dans cette commande, remplacez `3766c4ab331c` par l'ID du conteneur de l'étape précédente.

```
docker kill 3766c4ab331c
```

Déploiement de l'image

Pour charger l'image sur Amazon RIE et créer la fonction Lambda

1. Exécutez la get-login-passwordcommande pour authentifier la CLI Docker auprès de votre registre Amazon ECR.

- Définissez la `--region` valeur à l' Région AWS endroit où vous souhaitez créer le référentiel Amazon ECR.
- `111122223333`Remplacez-le par votre Compte AWS identifiant.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Créez un référentiel dans Amazon ECR à l'aide de la commande create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Le référentiel Amazon ECR doit être Région AWS identique à la fonction Lambda.

En cas de succès, vous obtenez une réponse comme celle-ci :

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Copiez le `repositoryUri` à partir de la sortie de l'étape précédente.

4. Exécutez la commande docker tag pour étiqueter votre image locale dans votre référentiel Amazon ECR en tant que dernière version. Dans cette commande :

   - `docker-image:test` est le nom et la balise de votre image Docker. Il s'agit du nom et de la balise de l'image que vous avez spécifiés dans la commande `docker build`.
   - Remplacez `<ECRrepositoryUri>` par l'`repositoryUri` que vous avez copié. Assurez-vous d'inclure `:latest` à la fin de l'URI.

   ```
   docker tag docker-image:test <ECRrepositoryUri>:latest
   ```

   Exemple :

   ```
   docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

5. Exécutez la commande docker push pour déployer votre image locale dans le référentiel Amazon ECR. Assurez-vous d'inclure `:latest` à la fin de l'URI du référentiel.

   ```
   docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

6. Créez un rôle d'exécution pour la fonction, si vous n'en avez pas déjà un. Vous aurez besoin de l'Amazon Resource Name (ARN) du rôle à l'étape suivante.
7. Créez la fonction Lambda. Pour `ImageUri`, indiquez l'URI du référentiel mentionné précédemment. Assurez-vous d'inclure `:latest` à la fin de l'URI.

   ```
   aws lambda create-function \
     --function-name hello-world \
     --package-type Image \
     --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
     --role arn:aws:iam::111122223333:role/lambda-ex
   ```

   > **Note**
   >
   > Vous pouvez créer une fonction en utilisant une image d'un autre AWS compte, à condition que l'image se trouve dans la même région que la fonction Lambda. Pour de plus amples informations, veuillez consulter Autorisations entre comptes Amazon ECR.

8. Invoquer la fonction.

```
aws lambda invoke --function-name hello-world response.json
```

Vous devriez obtenir une réponse comme celle-ci :

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Pour voir la sortie de la fonction, consultez le fichier `response.json`.

Pour mettre à jour le code de fonction, vous devez créer à nouveau l'image, télécharger la nouvelle image dans le référentiel Amazon ECR, puis utiliser la update-function-codecommande pour déployer l'image sur la fonction Lambda.

Lambda résout l'étiquette d'image en hachage d'image spécifique. Cela signifie que si vous pointez la balise d'image qui a été utilisée pour déployer la fonction vers une nouvelle image dans Amazon ECR, Lambda ne met pas automatiquement à jour la fonction pour utiliser la nouvelle image.

Pour déployer la nouvelle image sur la même fonction Lambda, vous devez utiliser la update-function-codecommande, même si la balise d'image dans Amazon ECR reste la même. Dans l'exemple suivant, l'option `--publish` crée une version de la fonction à l'aide de l'image du conteneur mise à jour.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Utilisation d'une autre image de base avec le client d'interface d'exécution

Si vous utilisez une image de base uniquement pour le système d'exploitation ou une autre image de base, vous devez inclure le client d'interface d'exécution dans votre image. Le client d'interface d'exécution étend leAPI de runtime, qui gère l'interaction entre Lambda et votre code de fonction.

Installez le client d'interface d'exécution pour Python à l'aide du gestionnaire de packages pip :

```
pip install awslambdaric
```

Vous pouvez également télécharger le client d'interface d'exécution Python depuis GitHub.

L'exemple suivant montre comment créer une image de conteneur pour Python à l'aide d'une image non AWS basique. L'exemple Dockerfile utilise une image de base Python officielle. Le Dockerfile inclut le client d'interface d'exécution pour Python.

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- AWS CLI version 2
- Docker (version minimale 25.0.0)
- Le plug-in buildx Docker.
- Python

Création d'une image à partir d'une image de base alternative

Pour créer une image de conteneur à partir d'une image non AWS basique

1. Créez un répertoire pour le projet, puis passez à ce répertoire.

   ```
   mkdir example
   cd example
   ```

2. Créez un nouveau fichier appelé `lambda_function.py`. Vous pouvez ajouter l'exemple de code de fonction suivant au fichier pour le tester, ou utiliser le vôtre.

   Example Fonction Python

   ```
   import sys
   def handler(event, context):
       return 'Hello from AWS Lambda using Python' + sys.version + '!'
   ```

3. Créez un nouveau fichier appelé `requirements.txt`. Si vous utilisez l'exemple de code de fonction de l'étape précédente, vous pouvez laisser le fichier vide, car il n'y a pas de dépendances. Sinon, répertoriez chaque bibliothèque requise. Par exemple, voici à quoi votre fichier `requirements.txt` doit ressembler si votre fonction utilise AWS SDK pour Python (Boto3) :

Example requirements.txt

```
boto3
```

4. Créez un nouveau fichier Docker. Le Dockerfile suivant utilise une image de base Python officielle au lieu d'une image de base AWS. Le Dockerfile inclut le client d'interface d'exécution, ce qui rend l'image compatible avec Lambda. L'exemple de fichier Docker suivant utilise une génération en plusieurs étapes.

   - Définissez la propriété `FROM` sur l'image de base.
   - Définissez le `ENTRYPOINT` sur le module que vous souhaitez que le conteneur Docker exécute lorsqu'il démarre. Dans ce cas, le module est le client d'interface d'exécution.
   - Définissez le `CMD` sur le gestionnaire de la fonction Lambda.

   Notez que l'exemple de Dockerfile n'inclut pas d'instruction USER. Lorsque vous déployez une image de conteneur sur Lambda, Lambda définit automatiquement un utilisateur Linux par défaut disposant d'autorisations de moindre privilège. Ceci est différent du comportement standard de Docker qui est défini par défaut par l'utilisateur `root` lorsqu'aucune instruction `USER` n'est fournie.

   Example Dockerfile

```
# Define custom function directory
ARG FUNCTION_DIR="/function"

FROM python:3.12 AS build-image

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Copy function code
RUN mkdir -p ${FUNCTION_DIR}
COPY . ${FUNCTION_DIR}

# Install the function's dependencies
RUN pip install \
    --target ${FUNCTION_DIR} \
        awslambdaric
```

```
# Use a slim version of the base Python image to reduce the final image size
FROM python:3.12-slim

# Include global arg in this stage of the build
ARG FUNCTION_DIR
# Set working directory to function root directory
WORKDIR ${FUNCTION_DIR}

# Copy in the built dependencies
COPY --from=build-image ${FUNCTION_DIR} ${FUNCTION_DIR}

# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "/usr/local/bin/python", "-m", "awslambdaric" ]
# Pass the name of the function handler as an argument to the runtime
CMD [ "lambda_function.handler" ]
```

5. Générez l'image Docker à l'aide de la commande docker build. L'exemple suivant nomme l'image `docker-image` et lui donne la balise `test`. Pour rendre votre image compatible avec Lambda, vous devez utiliser l'option `--provenance=false`.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> La commande spécifie l'option `--platform linux/amd64` pour garantir la compatibilité de votre conteneur avec l'environnement d'exécution Lambda, quelle que soit l'architecture de votre machine de génération. Si vous avez l'intention de créer une fonction Lambda à l'aide de l'architecture du jeu ARM64 d'instructions, veillez à modifier la commande pour utiliser l'`--platform linux/arm64`option à la place.

(Facultatif) Testez l'image localement

Utilisez l'émulateur d'interface d'exécution pour tester l'image localement. Vous pouvez intégrer l'émulateur dans votre image ou utiliser la procédure suivante pour l'installer sur votre machine locale.

Pour installer et exécuter l'émulateur d'interface d'exécution sur votre ordinateur local

1. Depuis le répertoire de votre projet, exécutez la commande suivante pour télécharger l'émulateur d'interface d'exécution (architecture x86-64) GitHub et l'installer sur votre machine locale.

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   Pour installer l'émulateur arm64, remplacez l'URL du GitHub référentiel dans la commande précédente par la suivante :

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

   PowerShell

   ```
   $dirPath = "$HOME\.aws-lambda-rie"
   if (-not (Test-Path $dirPath)) {
       New-Item -Path $dirPath -ItemType Directory
   }

   $downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie"
   $destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
   Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
   ```

   Pour installer l'émulateur arm64, remplacez `$downloadLink` par ce qui suit :

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

2. Démarrez votre image Docker à l'aide de la commande docker run. Notez ce qui suit :

   - `docker-image` est le nom de l'image et `test` est la balise.
   - `/usr/local/bin/python -m awslambdaric lambda_function.handler` est le `ENTRYPOINT` suivi du `CMD` depuis votre Dockerfile.

Linux/macOS

```
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
 9000:8080 \
    --entrypoint /aws-lambda/aws-lambda-rie \
    docker-image:test \
        /usr/local/bin/python -m awslambdaric lambda_function.handler
```

PowerShell

```
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
 9000:8080 `
--entrypoint /aws-lambda/aws-lambda-rie `
docker-image:test `
    /usr/local/bin/python -m awslambdaric lambda_function.handler
```

Cette commande exécute l'image en tant que conteneur et crée un point de terminaison local à `localhost:9000/2015-03-31/functions/function/invocations`.

> **Note**
>
> Si vous avez créé l'image Docker pour l'architecture du jeu ARM64 d'instructions, veillez à utiliser l'`--platform linux/arm64`option au lieu de`--platform linux/amd64`.

3. Publiez un événement au point de terminaison local.

Linux/macOS

Sous Linux et macOS, exécutez la commande `curl` suivante :

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

Cette commande invoque la fonction avec un événement vide et renvoie une réponse. Si vous utilisez votre propre code de fonction plutôt que l'exemple de code de fonction, vous pouvez invoquer la fonction avec une charge utile JSON. Exemple :

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

Dans PowerShell, exécutez la `Invoke-WebRequest` commande suivante :

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Cette commande invoque la fonction avec un événement vide et renvoie une réponse. Si vous utilisez votre propre code de fonction plutôt que l'exemple de code de fonction, vous pouvez invoquer la fonction avec une charge utile JSON. Exemple :

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. Obtenez l'ID du conteneur.

```
docker ps
```

5. Utilisez la commande docker kill pour arrêter le conteneur. Dans cette commande, remplacez `3766c4ab331c` par l'ID du conteneur de l'étape précédente.

```
docker kill 3766c4ab331c
```

Déploiement de l'image

Pour charger l'image sur Amazon RIE et créer la fonction Lambda

1. Exécutez la get-login-passwordcommande pour authentifier la CLI Docker auprès de votre registre Amazon ECR.
   - Définissez la `--region` valeur à l' Région AWS endroit où vous souhaitez créer le référentiel Amazon ECR.
   - `111122223333`Remplacez-le par votre Compte AWS identifiant.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Créez un référentiel dans Amazon ECR à l'aide de la commande create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Le référentiel Amazon ECR doit être Région AWS identique à la fonction Lambda.

En cas de succès, vous obtenez une réponse comme celle-ci :

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Copiez le `repositoryUri` à partir de la sortie de l'étape précédente.
4. Exécutez la commande docker tag pour étiqueter votre image locale dans votre référentiel Amazon ECR en tant que dernière version. Dans cette commande :

- `docker-image:test` est le nom et la balise de votre image Docker. Il s'agit du nom et de la balise de l'image que vous avez spécifiés dans la commande `docker build`.
- Remplacez `<ECRrepositoryUri>` par l'`repositoryUri` que vous avez copié. Assurez-vous d'inclure `:latest` à la fin de l'URI.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Exemple :

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Exécutez la commande docker push pour déployer votre image locale dans le référentiel Amazon ECR. Assurez-vous d'inclure `:latest` à la fin de l'URI du référentiel.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Créez un rôle d'exécution pour la fonction, si vous n'en avez pas déjà un. Vous aurez besoin de l'Amazon Resource Name (ARN) du rôle à l'étape suivante.
7. Créez la fonction Lambda. Pour `ImageUri`, indiquez l'URI du référentiel mentionné précédemment. Assurez-vous d'inclure `:latest` à la fin de l'URI.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> Vous pouvez créer une fonction en utilisant une image d'un autre AWS compte, à condition que l'image se trouve dans la même région que la fonction Lambda. Pour de plus amples informations, veuillez consulter Autorisations entre comptes Amazon ECR.

8. Invoquer la fonction.

```
aws lambda invoke --function-name hello-world response.json
```

Vous devriez obtenir une réponse comme celle-ci :

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Pour voir la sortie de la fonction, consultez le fichier `response.json`.

Pour mettre à jour le code de fonction, vous devez créer à nouveau l'image, télécharger la nouvelle image dans le référentiel Amazon ECR, puis utiliser la update-function-codecommande pour déployer l'image sur la fonction Lambda.

Lambda résout l'étiquette d'image en hachage d'image spécifique. Cela signifie que si vous pointez la balise d'image qui a été utilisée pour déployer la fonction vers une nouvelle image dans Amazon ECR, Lambda ne met pas automatiquement à jour la fonction pour utiliser la nouvelle image.

Pour déployer la nouvelle image sur la même fonction Lambda, vous devez utiliser la update-function-codecommande, même si la balise d'image dans Amazon ECR reste la même. Dans l'exemple suivant, l'option `--publish` crée une version de la fonction à l'aide de l'image du conteneur mise à jour.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Pour un exemple de création d'une image Python à partir d'une image de base Alpine, consultez Prise en charge des images conteneurs pour Lambda sur le blog AWS .

Utilisation de couches pour les fonctions Lambda Python

Utilisez les couches Lambda pour empaqueter le code et les dépendances que vous souhaitez réutiliser dans plusieurs fonctions. Les couches contiennent généralement des dépendances de bibliothèque, une exécution personnalisée, ou des fichiers de configuration. La création d'une couche implique trois étapes générales :

1. Empaquetez le contenu de votre couche. Cela signifie créer une archive de fichiers .zip contenant les dépendances que vous souhaitez utiliser dans vos fonctions.
2. Créez la couche dans Lambda.
3. Ajoutez la couche à vos fonctions.

Rubriques

- Empaqueter le contenu de votre couche
- Créer la couche dans Lambda
- Ajouter la couche à votre fonction
- Exemple d'application

Empaqueter le contenu de votre couche

Pour créer une couche, regroupez vos packages dans une archive (fichier .zip) répondant aux exigences suivantes :

- Créez la couche à l'aide de la même version de Python que celle que vous prévoyez d'utiliser pour la fonction Lambda. Par exemple, si vous créez votre couche à l'aide de Python 3.14, utilisez le runtime Python 3.14 pour votre fonction.
- Votre fichier .zip doit inclure un répertoire `python` au niveau racine.
- Les packages de votre couche doivent être compatibles avec Linux. Les fonctions Lambda s'exécutent sur Amazon Linux.

Vous pouvez créer des couches contenant des bibliothèques Python tierces installées avec `pip` (comme `requests` ou `pandas`) ou vos propres modules et packages Python.

Dépendances tierces

Créer une couche à l'aide de packages pip

1. Choisissez l'une des méthodes suivantes pour installer les packages `pip` dans le répertoire de niveau supérieur requis (`python/`) :

   pip install

   Pour les packages Python purs (comme requests ou boto3) :

   ```
   pip install requests -t python/
   ```

   Certains packages Python, tels que NumPy Pandas, incluent des composants C compilés. Si vous créez une couche avec ces packages sous macOS ou Windows, vous devrez peut-être utiliser cette commande pour installer un fichier wheel compatible avec Linux :

   ```
   pip install numpy --platform manylinux2014_x86_64 --only-binary=:all: -t python/
   ```

   Pour plus d'informations sur l'utilisation des packages Python contenant des composants compilés, consultez Création de packages de déploiement .zip avec des bibliothèques natives.

   requirements.txt

   L'utilisation d'un fichier `requirements.txt` vous permet de gérer les versions des packages et de garantir des installations cohérentes.

   Example requirements.txt

   ```
   requests==2.31.0
   boto3==1.37.34
   numpy==1.26.4
   ```

   Si votre fichier `requirements.txt` contient uniquement des packages Python purs (comme requests ou boto3) :

   ```
   pip install -r requirements.txt -t python/
   ```

Certains packages Python, tels que NumPy Pandas, incluent des composants C compilés. Si vous créez une couche avec ces packages sous macOS ou Windows, vous devrez peut-être utiliser cette commande pour installer un fichier wheel compatible avec Linux :

```
pip install -r requirements.txt --platform manylinux2014_x86_64 --only-
binary=:all: -t python/
```

Pour plus d'informations sur l'utilisation des packages Python contenant des composants compilés, consultez Création de packages de déploiement .zip avec des bibliothèques natives.

2. Compressez le contenu du répertoire `python` dans un fichier .zip.

Linux/macOS

```
zip -r layer.zip python/
```

PowerShell

```
Compress-Archive -Path .\python -DestinationPath .\layer.zip
```

La structure de répertoire de votre fichier .zip devrait se présenter comme suit :

```
python/              # Required top-level directory
### requests/
### boto3/
### numpy/
### (dependencies of the other packages)
```

> **Note**
>
> Si vous utilisez un environnement virtuel Python (venv) pour installer les packages, la structure de votre répertoire sera différente (par exemple, `python/lib/python3.x/site-packages`). Tant que votre fichier .zip inclut le répertoire `python` au niveau racine, Lambda peut localiser et importer vos packages.

Modules Python personnalisés

Créer une couche à l'aide de votre propre code

1. Créez le répertoire de niveau supérieur requis pour votre couche :

   ```
   mkdir python
   ```

2. Créez vos modules Python dans le répertoire `python`. L'exemple de module suivant valide les commandes en confirmant qu'elles contiennent les informations requises.

   Example module personnalisé : validator.py

   ```
   import json

   def validate_order(order_data):
       """Validates an order and returns formatted data."""
       required_fields = ['product_id', 'quantity']

       # Check required fields
       missing_fields = [field for field in required_fields if field not in
    order_data]
       if missing_fields:
           raise ValueError(f"Missing required fields: {', '.join(missing_fields)}")

       # Validate quantity
       quantity = order_data['quantity']
       if not isinstance(quantity, int) or quantity < 1:
           raise ValueError("Quantity must be a positive integer")

       # Format and return the validated data
       return {
           'product_id': str(order_data['product_id']),
           'quantity': quantity,
           'shipping_priority': order_data.get('priority', 'standard')
       }

   def format_response(status_code, body):
       """Formats the API response."""
       return {
           'statusCode': status_code,
           'body': json.dumps(body)
   ```

```
    }
```

3. Compressez le contenu du répertoire `python` dans un fichier .zip.

   Linux/macOS

   ```
   zip -r layer.zip python/
   ```

   PowerShell

   ```
   Compress-Archive -Path .\python -DestinationPath .\layer.zip
   ```

   La structure de répertoire de votre fichier .zip devrait se présenter comme suit :

   ```
   python/     # Required top-level directory
   ### validator.py
   ```

4. Dans votre fonction, importez et utilisez les modules comme vous le feriez avec n'importe quel package Python. Exemple :

   ```
   from validator import validate_order, format_response
   import json

   def lambda_handler(event, context):
       try:
           # Parse the order data from the event body
           order_data = json.loads(event.get('body', '{}'))

           # Validate and format the order
           validated_order = validate_order(order_data)

           return format_response(200, {
               'message': 'Order validated successfully',
               'order': validated_order
           })
       except ValueError as e:
           return format_response(400, {
               'error': str(e)
           })
       except Exception as e:
           return format_response(500, {
   ```

```
            'error': 'Internal server error'
        })
```

Vous pouvez utiliser l'événement de test suivant afin d'invoquer la fonction :

```
{
    "body": "{\"product_id\": \"ABC123\", \"quantity\": 2, \"priority\": \"express
\"}"
}
```

Réponse attendue :

```
{
  "statusCode": 200,
  "body": "{\"message\": \"Order validated successfully\", \"order\": {\"product_id
\": \"ABC123\", \"quantity\": 2, \"shipping_priority\": \"express\"}}"
}
```

Créer la couche dans Lambda

Vous pouvez publier votre couche à l'aide de la console Lambda AWS CLI ou de la console Lambda.

AWS CLI

Exécutez la publish-layer-version AWS CLI commande pour créer la couche Lambda :

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip
 --compatible-runtimes python3.14
```

Le paramètre compatible runtimes est facultatif. Lorsqu'il est spécifié, Lambda utilise ce paramètre pour filtrer les couches dans la console Lambda.

Console

Pour créer une couche (console)

1. Ouvrez la page Couches de la console Lambda.
2. Sélectionnez Créer un calque.
3. Choisissez Charger un fichier .zip, puis chargez l'archive .zip que vous avez créée précédemment.

4. (Facultatif) Pour Exécutions compatibles, choisissez l'environnement d'exécution Python qui correspond à la version de Python que vous avez utilisée pour créer votre couche.
5. Choisissez Créer.

Ajouter la couche à votre fonction

AWS CLI

Pour associer la couche à votre fonction, exécutez la update-function-configuration AWS CLI commande. Utilisez l'ARN de la couche pour le paramètre `--layers`. L'ARN doit spécifier la version (par exemple, `arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`). Pour de plus amples informations, veuillez consulter Couches et versions de couches.

```
aws lambda update-function-configuration --function-name my-function --cli-binary-format raw-in-base64-out --layers "arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1"
```

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

Console

Ajouter une couche à une fonction

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez la fonction.
3. Faites défiler jusqu'à la section Couches, puis choisissez Ajouter une couche.
4. Sous Choisir une couche, sélectionnez Couches personnalisées, puis choisissez votre couche.

> **Note**
>
> Si vous n'avez pas ajouté d'environnement d'exécution compatible lors de la création de la couche, celle-ci ne sera pas répertoriée ici. Vous pouvez spécifier l'ARN de la couche à la place.

5. Choisissez Ajouter.

Exemple d’application

Pour d'autres exemples d'utilisation des couches Lambda, consultez l'exemple d'application layer-python dans le référentiel du AWS Lambda Developer Guide. GitHub Cette application inclut deux couches contenant des bibliothèques Python. Après avoir créé les couches, vous pouvez déployer et invoquer les fonctions correspondantes pour vérifier que les couches fonctionnent comme prévu.

Utilisation de l'objet de contexte Lambda pour récupérer les informations de la fonction Python

Lorsque Lambda exécute votre fonction, il transmet un objet contexte au gestionnaire. Cet objet fournit des méthodes et des propriétés fournissant des informations sur l'appel, la fonction et l'environnement d'exécution. Pour en savoir plus sur la façon dont l'objet de contexte est transmis au gestionnaire de fonctions, consultez Définition du gestionnaire de fonction Lambda en Python.

Méthodes de contexte

- `get_remaining_time_in_millis` – Renvoie le nombre de millisecondes restant avant l'expiration de l'exécution.

Propriétés du contexte

- `function_name` – Nom de la fonction Lambda.
- `function_version` – Version de la fonction.
- `invoked_function_arn` – Amazon Resource Name (ARN) utilisé pour appeler la fonction. Indique si l'appelant a spécifié un numéro de version ou un alias.
- `memory_limit_in_mb` – Quantité de mémoire allouée à la fonction.
- `aws_request_id` – Identifiant de la demande d'invocation.
- `log_group_name` – Groupe de journaux pour la fonction.
- `log_stream_name` – Flux de journal de l'instance de fonction.
- `identity` – (applications mobiles) Informations sur l'identité Amazon Cognito qui a autorisé la demande.
  - `cognito_identity_id` – Identité Amazon Cognito authentifiée.
  - `cognito_identity_pool_id` – Groupe d'identités Amazon Cognito ayant autorisé l'invocation.
- `client_context` – (applications mobiles) Contexte client fourni à Lambda par l'application client.
  - `client.installation_id`
  - `client.app_title`
  - `client.app_version_name`
  - `client.app_version_code`
  - `client.app_package_name`

- `custom` – `dict` de valeurs personnalisées définies par l'application client mobile.
- `env` – `dict` d'informations d'environnement fournies par le kit SDK AWS.

Powertools pour Lambda (Python) fournit une définition d'interface pour l'objet de contexte Lambda. Vous pouvez utiliser la définition de l'interface pour les indications de type ou pour examiner plus en détail la structure de l'objet de contexte Lambda. Pour la définition de l'interface, consultez lambda_context.py dans le référentiel powertools-lambda-python sur GitHub.

L'exemple suivant montre une fonction de gestionnaire qui consigne les informations de contexte.

Example handler.py

```
import time

def lambda_handler(event, context):
    print("Lambda function ARN:", context.invoked_function_arn)
    print("CloudWatch log stream name:", context.log_stream_name)
    print("CloudWatch log group name:",  context.log_group_name)
    print("Lambda Request ID:", context.aws_request_id)
    print("Lambda function memory limits in MB:", context.memory_limit_in_mb)
    # We have added a 1 second delay so you can see the time remaining in
 get_remaining_time_in_millis.
    time.sleep(1)
    print("Lambda time remaining in MS:", context.get_remaining_time_in_millis())
```

En plus des options ci-dessus, vous pouvez également utiliser le kit SDK AWS X-Ray pour Instrumentation du code Python dans AWS Lambda afin d'identifier les chemins de code critiques, suivre leurs performances et capturer les données pour l'analyse.

Journalisation et surveillance des fonctions Lambda Python

AWS Lambda surveille automatiquement les fonctions Lambda et envoie des entrées de journal à Amazon. CloudWatch Votre fonction Lambda est fournie avec un groupe de CloudWatch journaux Logs et un flux de journaux pour chaque instance de votre fonction. L'environnement d'exécution Lambda envoie des détails sur chaque invocation et d'autres sorties provenant du code de votre fonction au flux de journaux. Pour plus d'informations sur CloudWatch les journaux, consultezEnvoi automatique des journaux de la fonction Lambda à CloudWatch Logs.

Pour produire des journaux à partir de votre code de fonction, vous pouvez utiliser le module intégré `logging`. Pour des entrées plus détaillées, vous pouvez utiliser n'importe quelle bibliothèque de journalisation qui écrit dans `stdout` ou `stderr`.

Impression dans le journal

Pour envoyer une sortie de base aux journaux, vous pouvez utiliser une méthode `print` dans votre fonction. L'exemple suivant enregistre les valeurs du groupe et du flux de CloudWatch journaux Logs, ainsi que l'objet de l'événement.

Notez que si votre fonction génère des journaux à l'aide d'`print`instructions Python, Lambda ne peut envoyer les résultats de journal à CloudWatch Logs qu'au format texte brut. Pour capturer des journaux au format JSON structuré, vous devez utiliser une bibliothèque de journalisation prise en charge. Pour plus d'informations, consultez the section called "Utilisation des contrôles de journalisation avancés de Lambda avec Python".

Example lambda_function.py

```
import os
def lambda_handler(event, context):
    print('## ENVIRONMENT VARIABLES')
    print(os.environ['AWS_LAMBDA_LOG_GROUP_NAME'])
    print(os.environ['AWS_LAMBDA_LOG_STREAM_NAME'])
    print('## EVENT')
    print(event)
```

Example sortie de journal

```
START RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95 Version: $LATEST
## ENVIRONMENT VARIABLES
/aws/lambda/my-function
```

```
2025/08/31/[$LATEST]3893xmpl7fac4485b47bb75b671a283c
## EVENT
{'key': 'value'}
END RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95
REPORT RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95  Duration: 15.74 ms  Billed
 Duration: 147 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 130.49 ms
XRAY TraceId: 1-5e34a614-10bdxmplf1fb44f07bc535a1   SegmentId: 07f5xmpl2d1f6f85
 Sampled: true
```

L'environnement d'exécution Python enregistre les lignes `START`, `END` et `REPORT` pour chaque invocation. La ligne `REPORT` comprend les données suivantes :

Champs de données de la ligne REPORT

- RequestId— L'identifiant de demande unique pour l'invocation.
- Duration – Temps que la méthode de gestion du gestionnaire de votre fonction a consacré au traitement de l'événement.
- Billed Duration : temps facturé pour l'invocation.
- Memory Size – Quantité de mémoire allouée à la fonction.
- Max Memory Used – Quantité de mémoire utilisée par la fonction. Lorsque les appels partagent un environnement d'exécution, Lambda indique la mémoire maximale utilisée pour toutes les invocations. Ce comportement peut entraîner une valeur signalée plus élevée que prévu.
- Init Duration : pour la première requête servie, temps qu'il a pris à l'exécution charger la fonction et exécuter le code en dehors de la méthode du gestionnaire.
- XRAY TraceId — Pour les demandes suivies, l'ID de AWS X-Ray trace.
- SegmentId— Pour les demandes tracées, l'identifiant du segment X-Ray.
- Sampled : pour les demandes suivies, résultat de l'échantillonnage.

Utilisation d'une bibliothèque de journalisation

Pour des journaux plus détaillés, utilisez le module de journalisation de la bibliothèque standard, ou de toute bibliothèque de journalisation tierce qui écrit à `stdout` ou `stderr`.

Pour les environnements d'exécution Python pris en charge, vous pouvez choisir si les journaux créés à l'aide du module `logging` standard sont capturés en texte brut ou en JSON. Pour en savoir plus, veuillez consulter la section the section called "Utilisation des contrôles de journalisation avancés de Lambda avec Python".

Actuellement, le format de journal par défaut pour tous les environnements d'exécution Python est le texte brut. L'exemple suivant montre comment les sorties de journal créées à l'aide du `logging` module standard sont capturées en texte brut dans CloudWatch Logs.

```
import os
import logging
logger = logging.getLogger()
logger.setLevel("INFO")

def lambda_handler(event, context):
    logger.info('## ENVIRONMENT VARIABLES')
    logger.info(os.environ['AWS_LAMBDA_LOG_GROUP_NAME'])
    logger.info(os.environ['AWS_LAMBDA_LOG_STREAM_NAME'])
    logger.info('## EVENT')
    logger.info(event)
```

La sortie de `logger` inclut le niveau de journal, l'horodatage et l'ID de demande.

```
START RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125 Version: $LATEST
[INFO]  2025-08-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ##
 ENVIRONMENT VARIABLES
[INFO]  2025-08-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    /aws/
lambda/my-function
[INFO]  2025-08-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    2025/01/31/
[$LATEST]1bbe51xmplb34a2788dbaa7433b0aa4d
[INFO]  2025-08-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ## EVENT
[INFO]  2025-08-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    {'key':
 'value'}
END RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125
REPORT RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125  Duration: 2.75 ms   Billed
 Duration: 117 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 113.51 ms
XRAY TraceId: 1-5e34a66a-474xmpl7c2534a87870b4370   SegmentId: 073cxmpl3e442861
 Sampled: true
```

> **Note**
>
> Lorsque le format de journal de votre fonction est défini sur du texte brut, le paramètre de niveau de journal par défaut pour les environnements d'exécution Python est WARN. Cela signifie que Lambda envoie uniquement des sorties de journal de niveau WARN ou inférieur à CloudWatch Logs. Pour modifier le niveau de journalisation par défaut, utilisez la méthode

Python `logging` `setLevel()` comme indiqué dans cet exemple de code. Si vous définissez le format de journal de votre fonction sur JSON, nous vous recommandons de configurer le niveau de journalisation de votre fonction à l'aide de Lambda Advanced Logging Controls et non en définissant le niveau de journalisation dans le code. Pour en savoir plus, consultez the section called "Utilisation du filtrage au niveau du journal avec Python"

Utilisation des contrôles de journalisation avancés de Lambda avec Python

Pour mieux contrôler la manière dont les journaux de vos fonctions sont capturés, traités et consommés, vous pouvez configurer les options de journalisation suivantes pour les environnements d'exécution Lambda Python pris en charge :

- Format de journal : choisissez entre le format texte brut et le format JSON structuré pour les journaux de votre fonction
- Niveau de journalisation : pour les journaux au format JSON, choisissez le niveau de détail des journaux que Lambda envoie à Amazon CloudWatch, par exemple ERROR, DEBUG ou INFO
- Groupe de journaux : choisissez le groupe de CloudWatch journaux auquel votre fonction envoie les journaux

Pour plus d'informations sur ces options de journalisation et pour savoir comment configurer votre fonction pour les utiliser, consultez the section called "Configuration de commandes de journalisation avancées pour votre fonction Lambda".

Pour en savoir plus sur l'utilisation du format de journal et les options de niveau de journal avec vos fonctions Python Lambda, consultez les instructions des sections suivantes.

Utilisation de journaux JSON structurés avec Python

Si vous sélectionnez JSON pour le format de journal de votre fonction, Lambda enverra les journaux produits par la bibliothèque de journalisation standard Python au CloudWatch format JSON structuré. Chaque objet de journal JSON contient au moins quatre paires clé-valeur avec les clés suivantes :

- `"timestamp"` - heure à laquelle le message de journal a été généré
- `"level"` - niveau de journalisation attribué au message
- `"message"` - contenu du message de journal
- `"requestId"` - identifiant unique de la demande pour l'invocation de la fonction

La bibliothèque Python `logging` peut également ajouter des paires clé-valeur supplémentaires, comme `"logger"`, dans cet objet JSON.

Les exemples présentés dans les sections suivantes montrent comment les sorties de journal générées à l'aide de la `logging` bibliothèque Python sont capturées dans CloudWatch Logs lorsque vous configurez le format de journal de votre fonction au format JSON.

Notez que si vous utilisez la méthode `print` pour produire des sorties de journal de base comme décrit dans the section called “Impression dans le journal”, Lambda capture ces sorties sous forme de texte brut, même si vous définissez le format de journalisation de votre fonction sur JSON.

Sorties JSON standard à l’aide de la bibliothèque de journalisation Python

Les exemples d'extrait de code et de sortie de journal suivants montrent comment les sorties de journal standard générées à l'aide de la `logging` bibliothèque Python sont capturées dans CloudWatch Logs lorsque le format de journal de votre fonction est défini sur JSON.

Example Code de journalisation Python

```
import logging
logger = logging.getLogger()

def lambda_handler(event, context):
    logger.info("Inside the handler function")
```

Example Enregistrement de journaux JSON

```
{
    "timestamp":"2025-10-27T19:17:45.586Z",
    "level":"INFO",
    "message":"Inside the handler function",
    "logger": "root",
    "requestId":"79b4f56e-95b1-4643-9700-2807f4e68189"
}
```

Enregistrement de paramètres supplémentaires au format JSON

Lorsque le journal de votre fonction est au format JSON, vous pouvez également journaliser des paramètres supplémentaires avec la bibliothèque `logging` Python standard en utilisant le mot-clé `extra` pour transmettre un dictionnaire Python à la sortie du journal.

Example Code de journalisation Python

```
import logging

def lambda_handler(event, context):
    logging.info(
        "extra parameters example",
        extra={"a":"b", "b": [3]},
    )
```

Example Enregistrement de journaux JSON

```
{
  "timestamp": "2025-11-02T15:26:28Z",
  "level": "INFO",
  "message": "extra parameters example",
  "logger": "root",
  "requestId": "3dbd5759-65f6-45f8-8d7d-5bdc79a3bd01",
  "a": "b",
  "b": [
    3
  ]
}
```

Exceptions de journalisation au format JSON

L'extrait de code suivant montre comment les exceptions Python sont capturées dans la sortie du journal de votre fonction lorsque vous définissez le format de journal sur JSON. Notez que les sorties de journal générées en utilisant `logging.exception` sont affectées au niveau de journal ERROR.

Example Code de journalisation Python

```
import logging

def lambda_handler(event, context):
    try:
        raise Exception("exception")
    except:
        logging.exception("msg")
```

Example Enregistrement de journaux JSON

```
{
  "timestamp": "2025-11-02T16:18:57Z",
  "level": "ERROR",
  "message": "msg",
  "logger": "root",
  "stackTrace": [
    "  File \"/var/task/lambda_function.py\", line 15, in lambda_handler\n    raise
 Exception(\"exception\")\n"
  ],
  "errorType": "Exception",
  "errorMessage": "exception",
  "requestId": "3f9d155c-0f09-46b7-bdf1-e91dab220855",
  "location": "/var/task/lambda_function.py:lambda_handler:17"
}
```

Journaux structurés JSON avec d'autres outils de journalisation

Si votre code utilise déjà une autre bibliothèque de journalisation, telle que Powertools for AWS Lambda, pour produire des journaux structurés en JSON, vous n'avez pas besoin d'apporter de modifications. AWS Lambda ne double code aucun journal déjà codé au format JSON. Même si vous configurez votre fonction pour utiliser le format de journal JSON, vos sorties de journalisation apparaissent CloudWatch dans la structure JSON que vous définissez.

L'exemple suivant montre comment les sorties de journal générées à l'aide du AWS Lambda package Powertools for sont capturées dans CloudWatch Logs. Le format de cette sortie de journal est le même, que la configuration de journalisation de votre fonction soit définie sur JSON ou TEXT. Pour plus d'informations sur l'utilisation de Powertools pour AWS Lambda, consultez the section called "Utilisation de Powertools pour AWS Lambda (Python) et AWS SAM pour la journalisation structurée" et the section called "Utilisation de Powertools pour AWS Lambda (Python) et AWS CDK pour la journalisation structurée"

Example Extrait de code de journalisation en Python (à l'aide de Powertools pour) AWS Lambda

```
from aws_lambda_powertools import Logger

logger = Logger()

def lambda_handler(event, context):
    logger.info("Inside the handler function")
```

Example Enregistrement de journal JSON (à l'aide de Powertools pour AWS Lambda)

```
{
    "level": "INFO",
    "location": "lambda_handler:7",
    "message": "Inside the handler function",
    "timestamp": "2025-10-31 22:38:21,010+0000",
    "service": "service_undefined",
    "xray_trace_id": "1-654181dc-65c15d6b0fecbdd1531ecb30"
}
```

Utilisation du filtrage au niveau du journal avec Python

En configurant le filtrage au niveau des journaux, vous pouvez choisir de n'envoyer que les journaux ayant un certain niveau de journalisation ou un niveau inférieur à CloudWatch Logs. Pour savoir comment configurer le filtrage de niveau journal de votre fonction, consultez the section called “Filtrage au niveau du journal”.

AWS Lambda Pour filtrer les journaux de votre application en fonction de leur niveau de journalisation, votre fonction doit utiliser des journaux au format JSON. Vous pouvez effectuer cette opération de deux façons :

- Créez des sorties de journal à l’aide de la bibliothèque Python `logging` standard et configurez votre fonction pour utiliser le format de journal JSON. AWS Lambda filtre ensuite les sorties de votre journal à l’aide de la paire clé-valeur « niveau » de l’objet JSON décrit dans the section called “Utilisation de journaux JSON structurés avec Python”. Pour savoir comment configurer le format de journal de votre fonction, consultez the section called “Configuration de commandes de journalisation avancées pour votre fonction Lambda”.
- Utilisez une autre bibliothèque ou méthode de journalisation pour créer des journaux structurés JSON dans votre code qui incluent une paire clé-valeur « niveau » définissant le niveau de sortie du journal. Par exemple, vous pouvez utiliser Powertools pour générer des sorties AWS Lambda de journal structurées JSON à partir de votre code.

  Vous pouvez également utiliser une instruction print pour générer un objet JSON contenant un identifiant de niveau de journalisation. L'instruction d'impression suivante produit une sortie au format JSON dans laquelle le niveau de journalisation est défini sur INFO. AWS Lambda enverra l'objet JSON à CloudWatch Logs si le niveau de journalisation de votre fonction est défini sur INFO, DEBUG ou TRACE.

```
print('{"msg":"My log message", "level":"info"}')
```

Pour que Lambda puisse filtrer les journaux de votre fonction, vous devez également inclure une paire `"timestamp"` clé-valeur dans la sortie de votre journal JSON. L'heure doit être spécifiée dans un format d'horodatage RFC 3339 valide. Si vous ne fournissez pas d'horodatage valide, Lambda attribuera au journal le niveau INFO et ajoutera un horodatage pour vous.

Affichage des journaux dans la console Lambda

Vous pouvez utiliser la console Lambda pour afficher la sortie du journal après avoir invoqué une fonction Lambda.

Si votre code peut être testé à partir de l'éditeur Code intégré, vous trouverez les journaux dans les résultats d'exécution. Lorsque vous utilisez la fonctionnalité de test de console pour invoquer une fonction, vous trouverez Sortie du journal dans la section Détails.

Afficher les journaux dans CloudWatch la console

Vous pouvez utiliser la CloudWatch console Amazon pour consulter les journaux de toutes les invocations de fonctions Lambda.

Pour afficher les journaux sur la CloudWatch console

1. Ouvrez la page Groupes de journaux sur la CloudWatch console.
2. Choisissez le groupe de journaux pour votre fonction (***your-function-name***/aws/lambda/).
3. Choisissez un flux de journaux.

Chaque flux de journal correspond à une instance de votre fonction. Un flux de journal apparaît lorsque vous mettez à jour votre fonction Lambda et lorsque des instances supplémentaires sont créées pour gérer les appels simultanés. Pour trouver les journaux d'un appel spécifique, nous vous recommandons d'instrumenter votre fonction avec. AWS X-Ray X-Ray enregistre des détails sur la demande et le flux de journaux dans le suivi.

Afficher les journaux avec AWS CLI

AWS CLI Il s'agit d'un outil open source qui vous permet d'interagir avec les AWS services à l'aide de commandes dans votre interface de ligne de commande. Pour effectuer les étapes de cette section, vous devez disposer de la version 2 de l'AWS CLI.

Vous pouvez utiliser AWS CLI pour récupérer les journaux d'une invocation à l'aide de l'option de commande `--log-type`. La réponse inclut un champ `LogResult` qui contient jusqu'à 4 Ko de journaux codés en base64 provenant de l'invocation.

Example récupérer un ID de journal

L'exemple suivant montre comment récupérer un ID de journal à partir du champ `LogResult` d'une fonction nommée `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Vous devriez voir la sortie suivante:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example décoder les journaux

Dans la même invite de commandes, utilisez l'utilitaire `base64` pour décoder les journaux. L'exemple suivant montre comment récupérer les journaux encodés en base64 pour `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

Vous devriez voir la sortie suivante :

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

L'utilitaire `base64` est disponible sous Linux, macOS et Ubuntu sous Windows. Les utilisateurs de macOS auront peut-être besoin d'utiliser `base64 -D`.

Example Script get-logs.sh

Dans la même invite de commandes, utilisez le script suivant pour télécharger les cinq derniers événements de journalisation. Le script utilise `sed` pour supprimer les guillemets du fichier de sortie et attend 15 secondes pour permettre la mise à disposition des journaux. La sortie comprend la réponse de Lambda, ainsi que la sortie de la commande `get-log-events`.

Copiez le contenu de l'exemple de code suivant et enregistrez-le dans votre répertoire de projet Lambda sous `get-logs.sh`.

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS et Linux (uniquement)

Dans la même invite de commandes, les utilisateurs macOS et Linux peuvent avoir besoin d'exécuter la commande suivante pour s'assurer que le script est exécutable.

```
chmod -R 755 get-logs.sh
```

Example récupérer les cinq derniers événements de journal

Dans la même invite de commande, exécutez le script suivant pour obtenir les cinq derniers événements de journalisation.

```
./get-logs.sh
```

Vous devriez voir la sortie suivante:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
```

```
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


Suppression de journaux

Les groupes de journaux ne sont pas supprimés automatiquement quand vous supprimez une fonction. Pour éviter de stocker des journaux indéfiniment, supprimez le groupe de journaux ou configurez une période de conservation à l'issue de laquelle les journaux sont supprimés automatiquement.

Utilisation d'autres outils et bibliothèques de journalisation

Powertools for AWS Lambda (Python) est une boîte à outils destinée aux développeurs qui permet de mettre en œuvre les meilleures pratiques sans serveur et d'accroître la rapidité des développeurs. L'utilitaire Logger fournit un enregistreur optimisé pour Lambda qui inclut des informations supplémentaires sur le contexte de fonction pour toutes vos fonctions avec une sortie structurée en JSON. Utilisez cet utilitaire pour effectuer les opérations suivantes :

- Capturer les champs clés du contexte Lambda, démarrer à froid et structurer la sortie de la journalisation sous forme de JSON
- Journaliser les événements d'invocation Lambda lorsque cela est demandé (désactivé par défaut)
- Imprimer tous les journaux uniquement pour un pourcentage d'invocations via l'échantillonnage des journaux (désactivé par défaut)
- Ajouter des clés supplémentaires au journal structuré à tout moment
- Utiliser un formateur de journaux personnalisé (Apportez votre propre formateur) pour produire des journaux dans une structure compatible avec le RFC de journalisation de votre organisation

Utilisation de Powertools pour AWS Lambda (Python) et AWS SAM pour la journalisation structurée

Suivez les étapes ci-dessous pour télécharger, créer et déployer un exemple d'application Hello World Python avec des modules Powertools for Python (français non garanti) intégrés à l'aide d' AWS SAM. Cette application met en œuvre un backend API de base et utilise Powertools pour émettre des journaux, des métriques et des traces. Elle se compose d'un point de terminaison Amazon API Gateway et d'une fonction Lambda. Lorsque vous envoyez une requête GET au point de terminaison API Gateway, la fonction Lambda appelle, envoie des journaux et des métriques au format métrique intégré à CloudWatch, et envoie des traces à. AWS X-Ray La fonction renvoie un message `hello world`.

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- Python 3.9
- AWS CLI version 2
- AWS SAM CLI version 1.75 ou ultérieure. Si vous disposez d'une ancienne version de la AWS SAM CLI, reportez-vous à la section Mise à niveau de la AWS SAM CLI.

Déployer un exemple d' AWS SAM application

1. Initialisez l'application à l'aide du modèle Hello World Python.

```
sam init --app-template hello-world-powertools-python --name sam-app --package-type
 Zip --runtime python3.9 --no-tracing
```

2. Créez l'application.

```
cd sam-app && sam build
```

3. Déployez l'application.

```
sam deploy --guided
```

4. Suivez les invites à l'écran. Appuyez sur `Enter` pour accepter les options par défaut fournies dans l'expérience interactive.

> **Note**
>
> Car l'autorisation n'a HelloWorldFunction peut-être pas été définie, est-ce que ça va ? , assurez-vous de participery.

5. Obtenez l'URL de l'application déployée :

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoquez le point de terminaison de l'API :

```
curl GET <URL_FROM_PREVIOUS_STEP>
```

En cas de succès, vous obtiendrez cette réponse :

```
{"message":"hello world"}
```

7. Pour obtenir les journaux de la fonction, exécutez sam logs. Pour en savoir plus, consultez Utilisation des journaux dans le Guide du développeur AWS Serverless Application Model .

```
sam logs --stack-name sam-app
```

La sortie du journal se présente comme suit :

```
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04
 2025-02-03T14:59:50.371000 INIT_START Runtime Version:
 python:3.9.v16    Runtime Version ARN: arn:aws:lambda:us-
east-1::runtime:07a48df201798d627f2b950f03bb227aab4a655a1d019c3296406f95937e2525
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.112000
 START RequestId: d455cfc4-7704-46df-901b-2a5cce9405be Version: $LATEST
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.114000 {
  "level": "INFO",
  "location": "hello:23",
  "message": "Hello world API - HTTP 200",
  "timestamp": "2025-02-03 14:59:51,113+0000",
  "service": "PowertoolsHelloWorld",
  "cold_start": true,
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "function_memory_size": "128",
```

```
  "function_arn": "arn:aws:lambda:us-east-1:111122223333:function:sam-app-
HelloWorldFunction-YBg8yfYtOc9j",
  "function_request_id": "d455cfc4-7704-46df-901b-2a5cce9405be",
  "correlation_id": "e73f8aef-5e07-436e-a30b-63e4b23f0047",
  "xray_trace_id": "1-63dd2166-434a12c22e1307ff2114f299"
}
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "function_name",
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "ColdStart",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "service": "PowertoolsHelloWorld",
  "ColdStart": [
    1.0
  ]
}
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "service"
          ]
        ],
```

```
        "Metrics": [
          {
            "Name": "HelloWorldInvocations",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "service": "PowertoolsHelloWorld",
  "HelloWorldInvocations": [
    1.0
  ]
}
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.128000 END
 RequestId: d455cfc4-7704-46df-901b-2a5cce9405be
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.128000
 REPORT RequestId: d455cfc4-7704-46df-901b-2a5cce9405be    Duration: 16.33 ms
 Billed Duration: 756 ms    Memory Size: 128 MB    Max Memory Used: 64 MB    Init
 Duration: 739.46 ms
XRAY TraceId: 1-63dd2166-434a12c22e1307ff2114f299    SegmentId: 3c5d18d735a1ced0
 Sampled: true
```

8. Il s’agit d’un point de terminaison d’API public accessible par Internet. Nous vous recommandons de supprimer le point de terminaison après un test.

```
sam delete
```

Gestion de la conservation des journaux

Les groupes de journaux ne sont pas supprimés automatiquement quand vous supprimez une fonction. Pour éviter de stocker les journaux indéfiniment, supprimez le groupe de journaux ou configurez une période de conservation après laquelle les journaux CloudWatch sont automatiquement supprimés. Pour configurer la conservation des journaux, ajoutez ce qui suit à votre AWS SAM modèle :

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties
```

```
  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Utilisation de Powertools pour AWS Lambda (Python) et AWS CDK pour la journalisation structurée

Suivez les étapes ci-dessous pour télécharger, créer et déployer un exemple d'application Hello World Python avec des modules Powertools pour AWS Lambda (Python) intégrés à l'aide du AWS CDK. Cette application met en œuvre un backend API de base et utilise Powertools pour émettre des journaux, des métriques et des traces. Elle se compose d'un point de terminaison Amazon API Gateway et d'une fonction Lambda. Lorsque vous envoyez une requête GET au point de terminaison API Gateway, la fonction Lambda appelle, envoie des journaux et des métriques au format métrique intégré à CloudWatch, et envoie des traces à. AWS X-Ray La fonction renvoie un message hello world.

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- Python 3.9
- AWS CLI version 2
- AWS CDK version 2
- AWS SAM CLI version 1.75 ou ultérieure. Si vous disposez d'une ancienne version de la AWS SAM CLI, reportez-vous à la section Mise à niveau de la AWS SAM CLI.

Déployer un exemple d' AWS CDK application

1. Créez un répertoire de projets pour votre nouvelle application.

   ```
   mkdir hello-world
   cd hello-world
   ```

2. Initialisez l'application.

```
cdk init app --language python
```

3. Installez les dépendances de Python.

```
pip install -r requirements.txt
```

4. Créez un répertoire lambda_function dans le dossier racine.

```
mkdir lambda_function
cd lambda_function
```

5. Créez un fichier app.py et ajoutez-y le code suivant. Il s'agit du code de la fonction Lambda.

```
from aws_lambda_powertools.event_handler import APIGatewayRestResolver
from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools.logging import correlation_paths
from aws_lambda_powertools import Logger
from aws_lambda_powertools import Tracer
from aws_lambda_powertools import Metrics
from aws_lambda_powertools.metrics import MetricUnit

app = APIGatewayRestResolver()
tracer = Tracer()
logger = Logger()
metrics = Metrics(namespace="PowertoolsSample")

@app.get("/hello")
@tracer.capture_method
def hello():
    # adding custom metrics
    # See: https://docs.aws.amazon.com/powertools/python/latest//latest/core/
metrics/
    metrics.add_metric(name="HelloWorldInvocations", unit=MetricUnit.Count,
 value=1)

    # structured log
    # See: https://docs.aws.amazon.com/powertools/python/latest//latest/core/
logger/
    logger.info("Hello world API - HTTP 200")
    return {"message": "hello world"}

# Enrich logging with contextual information from Lambda
```

```
@logger.inject_lambda_context(correlation_id_path=correlation_paths.API_GATEWAY_REST)
# Adding tracer
# See: https://docs.aws.amazon.com/powertools/python/latest//latest/core/tracer/
@tracer.capture_lambda_handler
# ensures metrics are flushed upon request completion/failure and capturing
 ColdStart metric
@metrics.log_metrics(capture_cold_start_metric=True)
def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)
```

6. Ouvrez le répertoire hello_world. Vous devriez voir un fichier nommé hello_world_stack.py.

```
cd ..
cd hello_world
```

7. Ouvrez hello_world_stack.py et ajoutez le code suivant au fichier. Il contient le constructeur Lambda, qui crée la fonction Lambda, configure les variables d'environnement pour Powertools et fixe la durée de conservation des journaux à une semaine, et le constructeur ApiGatewayv 1, qui crée l'API REST.

```
from aws_cdk import (
    Stack,
    aws_apigateway as apigwv1,
    aws_lambda as lambda_,
    CfnOutput,
    Duration
)
from constructs import Construct

class HelloWorldStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # Powertools Lambda Layer
        powertools_layer = lambda_.LayerVersion.from_layer_version_arn(
            self,
            id="lambda-powertools",
            # Using AWS Lambda Powertools via Lambda Layer
            # This imports the Powertools layer which provides observability
 features for Lambda functions
            # For available versions, see: https://docs.aws.amazon.com/powertools/
python/latest/#lambda-layer
```

```
        )

        function = lambda_.Function(self,
            'sample-app-lambda',
            runtime=lambda_.Runtime.PYTHON_3_9,
            layers=[powertools_layer],
            code = lambda_.Code.from_asset("./lambda_function/"),
            handler="app.lambda_handler",
            memory_size=128,
            timeout=Duration.seconds(3),
            architecture=lambda_.Architecture.X86_64,
            environment={
                "POWERTOOLS_SERVICE_NAME": "PowertoolsHelloWorld",
                "POWERTOOLS_METRICS_NAMESPACE": "PowertoolsSample",
                "LOG_LEVEL": "INFO"
            }
        )

        apigw = apigwv1.RestApi(self, "PowertoolsAPI",
 deploy_options=apigwv1.StageOptions(stage_name="dev"))

        hello_api = apigw.root.add_resource("hello")
        hello_api.add_method("GET", apigwv1.LambdaIntegration(function,
 proxy=True))

        CfnOutput(self, "apiUrl", value=f"{apigw.url}hello")
```

8. Déployez votre application.

```
cd ..
cdk deploy
```

9. Obtenez l'URL de l'application déployée :

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`apiUrl`].OutputValue' --output text
```

10. Invoquez le point de terminaison de l'API :

```
curl GET <URL_FROM_PREVIOUS_STEP>
```

En cas de succès, vous obtiendrez cette réponse :

```
{"message":"hello world"}
```

11. Pour obtenir les journaux de la fonction, exécutez sam logs. Pour en savoir plus, consultez Utilisation des journaux dans le Guide du développeur AWS Serverless Application Model .

```
sam logs --stack-name HelloWorldStack
```

La sortie du journal se présente comme suit :

```
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04
 2025-02-03T14:59:50.371000 INIT_START Runtime Version:
 python:3.9.v16    Runtime Version ARN: arn:aws:lambda:us-
east-1::runtime:07a48df201798d627f2b950f03bb227aab4a655a1d019c3296406f95937e2525
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.112000
 START RequestId: d455cfc4-7704-46df-901b-2a5cce9405be Version: $LATEST
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.114000 {
  "level": "INFO",
  "location": "hello:23",
    "message": "Hello world API - HTTP 200",
  "timestamp": "2025-02-03 14:59:51,113+0000",
  "service": "PowertoolsHelloWorld",
  "cold_start": true,
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "function_memory_size": "128",
  "function_arn": "arn:aws:lambda:us-east-1:111122223333:function:sam-app-
HelloWorldFunction-YBg8yfYtOc9j",
  "function_request_id": "d455cfc4-7704-46df-901b-2a5cce9405be",
  "correlation_id": "e73f8aef-5e07-436e-a30b-63e4b23f0047",
  "xray_trace_id": "1-63dd2166-434a12c22e1307ff2114f299"
}
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "function_name",
            "service"
          ]
```

```
        ],
        "Metrics": [
          {
            "Name": "ColdStart",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "service": "PowertoolsHelloWorld",
  "ColdStart": [
    1.0
  ]
}
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "HelloWorldInvocations",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "service": "PowertoolsHelloWorld",
  "HelloWorldInvocations": [
    1.0
  ]
}
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.128000 END
 RequestId: d455cfc4-7704-46df-901b-2a5cce9405be
```

```
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.128000
 REPORT RequestId: d455cfc4-7704-46df-901b-2a5cce9405be    Duration: 16.33 ms
 Billed Duration: 756 ms    Memory Size: 128 MB    Max Memory Used: 64 MB    Init
 Duration: 739.46 ms
XRAY TraceId: 1-63dd2166-434a12c22e1307ff2114f299    SegmentId: 3c5d18d735a1ced0
 Sampled: true
```

12. Il s’agit d’un point de terminaison d’API public accessible par Internet. Nous vous recommandons de supprimer le point de terminaison après un test.

```
cdk destroy
```

Test de la fonction AWS Lambda dans Python

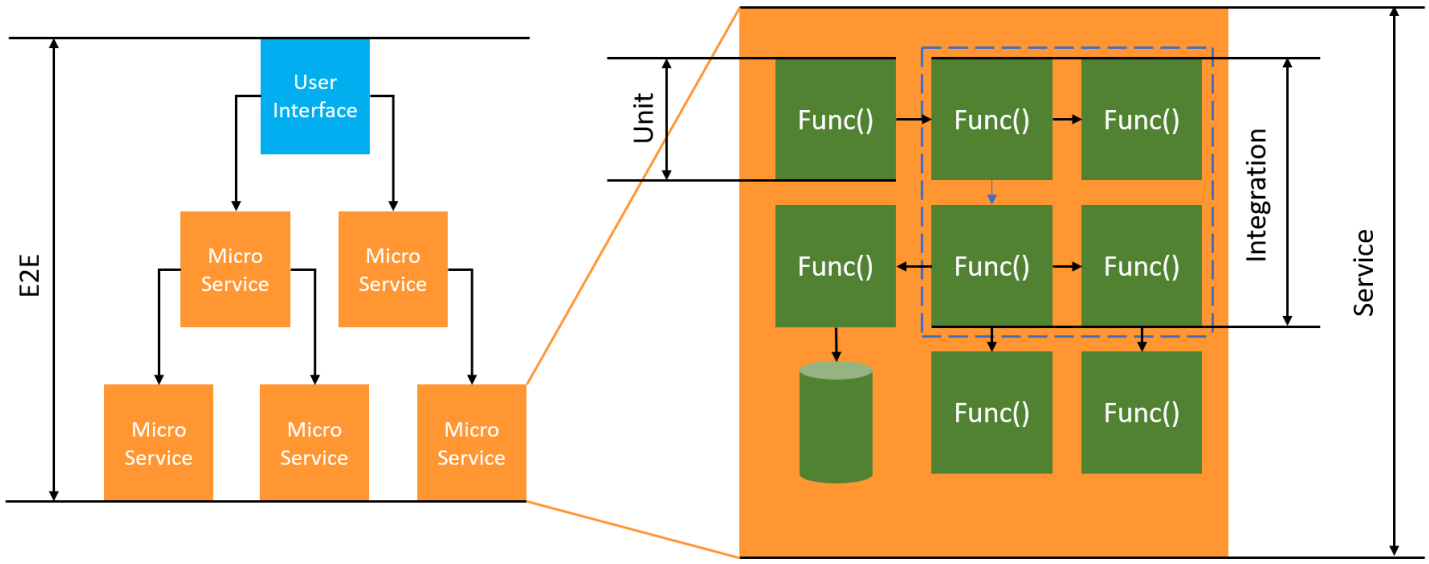
> **Note**
>
> Consultez le chapitre Test des fonctions pour une présentation complète des techniques et des bonnes pratiques pour tester les solutions sans serveur.

Le test des fonctions sans serveur utilise les types et techniques de tests traditionnels, mais vous devez également envisager de tester les applications sans serveur dans leur ensemble. Les tests basés sur le cloud fourniront la mesure la plus précise de la qualité de vos fonctions et de vos applications sans serveur.

Une architecture d'application sans serveur comprend des services gérés qui fournissent des fonctionnalités d'applications critiques par le biais d'appels d'API. C'est pourquoi votre cycle de développement doit inclure des tests automatisés qui vérifient la fonctionnalité lorsque votre fonction et vos services interagissent.

Si vous ne créez pas de tests basés sur le cloud, vous pouvez rencontrer des problèmes en raison des différences entre votre environnement local et l'environnement déployé. Votre processus d'intégration continue doit exécuter des tests sur une série de ressources allouées dans le cloud avant de promouvoir votre code vers l'environnement de déploiement suivant, comme l'assurance qualité, la mise en place ou la production.

Poursuivez la lecture de ce petit guide pour en savoir plus sur les stratégies de test pour les applications sans serveur, ou rendez-vous sur le référentiel Serverless Test Samples pour vous plonger dans des exemples pratiques, spécifiques au langage et à l'exécution que vous avez choisis.



Pour les tests sans serveur, vous écrirez toujours des tests unitaires, d'intégration et de bout en bout.

- Tests unitaires : tests exécutés sur un bloc de code isolé. Par exemple, la vérification de la logique métier permettant de calculer les frais de livraison en fonction d'un article et d'une destination donnés.
- Tests d'intégration : tests impliquant au moins deux composants ou services qui interagissent, généralement dans un environnement cloud. Par exemple, la vérification d'une fonction traite les événements d'une file d'attente.
- Tests de bout en bout : tests qui vérifient le comportement de l'ensemble d'une application. Il s'agit par exemple de s'assurer que l'infrastructure est correctement mise en place et que les événements circulent entre les services comme prévu pour enregistrer la commande d'un client.

Test de vos applications sans serveur

Vous utiliserez généralement une combinaison d'approches pour tester le code de votre application sans serveur, notamment des tests dans le cloud, des tests avec des simulations et, occasionnellement, des tests avec des émulateurs.

Tests dans le cloud

Les tests dans le cloud sont utiles pour toutes les phases des tests, y compris les tests unitaires, les tests d'intégration et les tests de bout en bout. Vous exécutez des tests sur du code déployé dans le cloud et interagissez avec des services basés sur le cloud. Cette approche fournit la mesure la plus précise de la qualité de votre code.

Un moyen pratique de déboguer votre fonction Lambda dans le cloud est de passer par la console avec un événement de test. Un événement de test est une entrée JSON pour votre fonction. Si votre fonction ne nécessite pas d'entrée, l'événement peut être un document JSON vide (`{}`). La console fournit des exemples d'événements pour diverses intégrations de services. Après avoir créé un événement dans la console, vous pouvez le partager avec votre équipe pour faciliter les tests et les rendre plus cohérents.

> **Note**
>
> Tester une fonction dans la console est un moyen rapide de démarrer, mais l'automatisation de vos cycles de test garantit la qualité des applications et la rapidité du développement.

Outils de test

Il existe des outils et des techniques permettant d'accélérer les boucles de rétroaction du développement. Par exemple, AWS SAM Accelerate et le mode de surveillance AWS CDK réduisent tous deux le temps nécessaire à la mise à jour des environnements cloud.

Moto est une bibliothèque Python qui permet de simuler des services et des ressources AWS. Vous pouvez ainsi tester vos fonctions avec peu ou pas de modifications à l'aide de décorateurs pour intercepter et simuler des réponses.

La fonctionnalité de validation de Powertools pour AWS Lambda (Python) fournit des décorateurs qui vous permettent de valider les événements d'entrée et les réponses de sortie de vos fonctions Python.

Pour plus d'informations, consultez l'article de blog Unit Testing Lambda with Python and Mock AWS Services.

Pour réduire le temps de latence liés aux itérations de déploiement dans le cloud, consultez AWS Serverless Application Model (AWS SAM) et le mode de surveillance AWS Cloud Development Kit (AWS CDK). Ces outils surveillent les modifications apportées à votre infrastructure et à votre code. Ils réagissent à ces modifications en créant et en déployant automatiquement des mises à jour incrémentielles dans votre environnement cloud.

Des exemples utilisant ces outils sont disponibles dans le référentiel de code Python Test Samples.

Instrumentation du code Python dans AWS Lambda

Lambda s'intègre avec AWS X-Ray pour vous aider à suivre, déboguer et optimiser les applications Lambda. Vous pouvez utiliser X-Ray pour suivre une demande lorsque celle-ci parcourt les ressources de votre application, qui peuvent inclure des fonctions Lambda et d'autres services AWS.

Pour envoyer des données de traçage à X-Ray, vous pouvez utiliser l'une des trois bibliothèques SDK suivantes :

- AWS Distro de OpenTelemetry (ADOT) : solution sécurisée, prête pour la production, distribution prise en charge par AWS du kit SDK OpenTelemetry (OTel).
- Kit SDK AWS X-Ray pour Python – Un kit SDK permettant de générer et d'envoyer des données de suivi à X-Ray.
- Powertools pour AWS Lambda (Python) : une boîte à outils pour développeurs permettant de mettre en œuvre les bonnes pratiques en matière de sans serveur et d'augmenter la rapidité des développeurs.

Chacun des kits SDK offre des moyens d'envoyer vos données de télémétrie au service X-Ray. Vous pouvez ensuite utiliser X-Ray pour afficher, filtrer et avoir un aperçu des métriques de performance de votre application, afin d'identifier les problèmes et les occasions d'optimiser votre application.

> **Important**
>
> X-Ray et Powertools pour les kits SDK AWS Lambda font partie d'une solution d'instrumentation étroitement intégrée proposée par AWS. Les couches ADOT Lambda font partie d'une norme industrielle pour l'instrumentation de traçage qui collecte plus de données en général, mais qui peut ne pas convenir à tous les cas d'utilisation. Vous pouvez mettre en œuvre le traçage de bout en bout dans X-Ray en utilisant l'une ou l'autre solution. Pour en savoir plus sur le choix entre les deux, consultez Choix entre les kits SDK AWS Distro for OpenTelemetry et X-Ray (français non garanti).

Sections

- Utilisation de Powertools pour AWS Lambda (Python) et d'AWS SAM pour le traçage
- Utilisation de Powertools pour AWS Lambda (Python) et de AWS CDK pour le traçage
- Utilisation d'ADOT pour instrumenter vos fonctions python

- Utilisation du kit SDK X-Ray pour instrumenter vos fonctions Python
- Activation du suivi avec la console Lambda
- Activation du suivi avec l'API Lambda
- Activation du suivi avec CloudFormation
- Interprétation d'un suivi X-Ray
- Stockage des dépendances d'exécution dans une couche (kit SDK X-Ray)

Utilisation de Powertools pour AWS Lambda (Python) et d'AWS SAM pour le traçage

Suivez les étapes ci-dessous pour télécharger, créer et déployer un exemple d'application Hello World Python avec des modules Powertools pour AWS Lambda (Python) intégrés à l'aide d'AWS SAM. Cette application met en œuvre un backend API de base et utilise Powertools pour émettre des journaux, des métriques et des traces. Elle se compose d'un point de terminaison Amazon API Gateway et d'une fonction Lambda. Lorsque vous envoyez une demande GET au point de terminaison API Gateway, la fonction Lambda est invoquée, envoie des journaux et des métriques à l'aide du format Embedded Metric à CloudWatch, et envoie des traces à AWS X-Ray. La fonction renvoie un message hello world.

Prérequis

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- Python 3.11
- AWS CLI Version 2 de 2
- AWS SAM CLI version 1.75 ou ultérieure. Si vous avez une version antérieure de la CLI AWS SAM, consultez Mise à niveau de la CLI AWS SAM.

Déployer un exemple d'application AWS SAM

1. Initialisez l'application à l'aide du modèle Hello World Python.

```
sam init --app-template hello-world-powertools-python --name sam-app --package-type
 Zip --runtime python3.11 --no-tracing
```

2. Créez l'application.

```
cd sam-app && sam build
```

3. Déployez l'application.

```
sam deploy --guided
```

4. Suivez les invites à l'écran. Appuyez sur `Enter` pour accepter les options par défaut fournies dans l'expérience interactive.

> **Note**
>
> Pour HelloWorldFunction n'a peut-être pas d'autorisation définie, est-ce correct ?, assurez-vous de saisir y.

5. Obtenez l'URL de l'application déployée :

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoquez le point de terminaison de l'API :

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

En cas de succès, vous obtiendrez cette réponse :

```
{"message":"hello world"}
```

7. Pour obtenir les traces de la fonction, exécutez sam traces.

```
sam traces
```

La sortie de la trace ressemble à ceci :

```
New XRay Service Graph
  Start time: 2023-02-03 14:59:50+00:00
  End time: 2023-02-03 14:59:50+00:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-YBg8yfYtOc9j -
 Edges: [1]
```

```
    Summary_statistics:
      - total requests: 1
      - ok count(2XX): 1
      - error count(4XX): 0
      - fault count(5XX): 0
      - total response time: 0.924
   Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-YBg8yfYtOc9j
  - Edges: []
    Summary_statistics:
      - total requests: 1
      - ok count(2XX): 1
      - error count(4XX): 0
      - fault count(5XX): 0
      - total response time: 0.016
   Reference Id: 2 - client - sam-app-HelloWorldFunction-YBg8yfYtOc9j - Edges: [0]
    Summary_statistics:
      - total requests: 0
      - ok count(2XX): 0
      - error count(4XX): 0
      - fault count(5XX): 0
      - total response time: 0

XRay Event [revision 1] at (2023-02-03T14:59:50.204000) with id
 (1-63dd2166-434a12c22e1307ff2114f299) and duration (0.924s)
 - 0.924s - sam-app-HelloWorldFunction-YBg8yfYtOc9j [HTTP: 200]
 - 0.016s - sam-app-HelloWorldFunction-YBg8yfYtOc9j
   - 0.739s - Initialization
   - 0.016s - Invocation
     - 0.013s - ## lambda_handler
       - 0.000s - ## app.hello
   - 0.000s - Overhead
```

8. Il s’agit d’un point de terminaison d’API public accessible par Internet. Nous vous recommandons de supprimer le point de terminaison après un test.

```
sam delete
```

X-Ray ne trace pas toutes les requêtes vers votre application. X-Ray applique un algorithme d’échantillonnage pour s’assurer que le suivi est efficace, tout en fournissant un échantillon représentatif de toutes les demandes. Le taux d’échantillonnage est 1 demande par seconde et 5 %

de demandes supplémentaires. Vous ne pouvez pas configurer ce taux d'échantillonnage X-Ray pour vos fonctions.

Utilisation de Powertools pour AWS Lambda (Python) et de AWS CDK pour le traçage

Suivez les étapes ci-dessous pour télécharger, créer et déployer un exemple d'application Hello World Python avec des modules Powertools pour AWS Lambda (Python) intégrés à l'aide d'AWS CDK. Cette application met en œuvre un backend API de base et utilise Powertools pour émettre des journaux, des métriques et des traces. Elle se compose d'un point de terminaison Amazon API Gateway et d'une fonction Lambda. Lorsque vous envoyez une demande GET au point de terminaison API Gateway, la fonction Lambda est invoquée, envoie des journaux et des métriques à l'aide du format Embedded Metric à CloudWatch, et envoie des traces à AWS X-Ray. La fonction renvoie un message hello world.

Prérequis

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- Python 3.11
- AWS CLI Version 2 de 2
- AWS CDK Version 2 de 2
- CLI AWS SAM version 1.75 ou ultérieure. Si vous avez une version antérieure de la CLI AWS SAM, consultez Mise à niveau de la CLI AWS SAM.

Déployer un exemple d'application AWS CDK

1. Créez un répertoire de projets pour votre nouvelle application.

```
mkdir hello-world
cd hello-world
```

2. Initialisez l'application.

```
cdk init app --language python
```

3. Installez les dépendances de Python.

```
pip install -r requirements.txt
```

4. Créez un répertoire lambda_function dans le dossier racine.

```
mkdir lambda_function
cd lambda_function
```

5. Créez un fichier app.py et ajoutez-y le code suivant. Il s'agit du code de la fonction Lambda.

```
from aws_lambda_powertools.event_handler import APIGatewayRestResolver
from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools.logging import correlation_paths
from aws_lambda_powertools import Logger
from aws_lambda_powertools import Tracer
from aws_lambda_powertools import Metrics
from aws_lambda_powertools.metrics import MetricUnit

app = APIGatewayRestResolver()
tracer = Tracer()
logger = Logger()
metrics = Metrics(namespace="PowertoolsSample")

@app.get("/hello")
@tracer.capture_method
def hello():
    # adding custom metrics
    # See: https://docs.powertools.aws.dev/lambda-python/latest/core/metrics/
    metrics.add_metric(name="HelloWorldInvocations", unit=MetricUnit.Count,
 value=1)

    # structured log
    # See: https://docs.powertools.aws.dev/lambda-python/latest/core/logger/
    logger.info("Hello world API - HTTP 200")
    return {"message": "hello world"}

# Enrich logging with contextual information from Lambda
@logger.inject_lambda_context(correlation_id_path=correlation_paths.API_GATEWAY_REST)
# Adding tracer
# See: https://docs.powertools.aws.dev/lambda-python/latest/core/tracer/
@tracer.capture_lambda_handler
# ensures metrics are flushed upon request completion/failure and capturing
 ColdStart metric
```

```
@metrics.log_metrics(capture_cold_start_metric=True)
def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)
```

6. Ouvrez le répertoire hello_world. Vous devriez voir un fichier nommé hello_world_stack.py.

```
cd ..
cd hello_world
```

7. Ouvrez hello_world_stack.py et ajoutez le code suivant au fichier. Celui-ci contient le Lambda Constructor, qui crée la fonction Lambda, configure les variables d'environnement pour Powertools et définit la rétention des journaux à une semaine, ainsi que l'ApiGatewayv1 Constructor, qui crée l'API REST.

```
from aws_cdk import (
    Stack,
    aws_apigateway as apigwv1,
    aws_lambda as lambda_,
    CfnOutput,
    Duration
)
from constructs import Construct

class HelloWorldStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # Powertools Lambda Layer
        powertools_layer = lambda_.LayerVersion.from_layer_version_arn(
            self,
            id="lambda-powertools",
            # At the moment we wrote this example, the aws_lambda_python_alpha CDK
 constructor is in Alpha, o we use layer to make the example simpler
            # See https://docs.aws.amazon.com/cdk/api/v2/python/
aws_cdk.aws_lambda_python_alpha/README.html
            # Check all Powertools layers versions here: https://
docs.powertools.aws.dev/lambda-python/latest/#lambda-layer
            layer_version_arn=f"arn:aws:lambda:
{self.region}:017000801446:layer:AWSLambdaPowertoolsPythonV2:21"
        )

        function = lambda_.Function(self,
```

```
            'sample-app-lambda',
            runtime=lambda_.Runtime.PYTHON_3_11,
            layers=[powertools_layer],
            code = lambda_.Code.from_asset("./lambda_function/"),
            handler="app.lambda_handler",
            memory_size=128,
            timeout=Duration.seconds(3),
            architecture=lambda_.Architecture.X86_64,
            environment={
                "POWERTOOLS_SERVICE_NAME": "PowertoolsHelloWorld",
                "POWERTOOLS_METRICS_NAMESPACE": "PowertoolsSample",
                "LOG_LEVEL": "INFO"
            }
        )

        apigw = apigwv1.RestApi(self, "PowertoolsAPI",
 deploy_options=apigwv1.StageOptions(stage_name="dev"))

        hello_api = apigw.root.add_resource("hello")
        hello_api.add_method("GET", apigwv1.LambdaIntegration(function,
 proxy=True))

        CfnOutput(self, "apiUrl", value=f"{apigw.url}hello")
```

8. Déployez votre application.

```
cd ..
cdk deploy
```

9. Obtenez l'URL de l'application déployée :

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`apiUrl`].OutputValue' --output text
```

10. Invoquez le point de terminaison de l'API :

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

En cas de succès, vous obtiendrez cette réponse :

```
{"message":"hello world"}
```

11. Pour obtenir les traces de la fonction, exécutez sam traces.

```
sam traces
```

La sortie des traces ressemble à ceci :

```
New XRay Service Graph
  Start time: 2023-02-03 14:59:50+00:00
  End time: 2023-02-03 14:59:50+00:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-YBg8yfYtOc9j -
 Edges: [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.924
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-YBg8yfYtOc9j
 - Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.016
  Reference Id: 2 - client - sam-app-HelloWorldFunction-YBg8yfYtOc9j - Edges: [0]
   Summary_statistics:
     - total requests: 0
     - ok count(2XX): 0
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0

XRay Event [revision 1] at (2023-02-03T14:59:50.204000) with id
 (1-63dd2166-434a12c22e1307ff2114f299) and duration (0.924s)
 - 0.924s - sam-app-HelloWorldFunction-YBg8yfYtOc9j [HTTP: 200]
 - 0.016s - sam-app-HelloWorldFunction-YBg8yfYtOc9j
   - 0.739s - Initialization
   - 0.016s - Invocation
     - 0.013s - ## lambda_handler
       - 0.000s - ## app.hello
   - 0.000s - Overhead
```

12. Il s'agit d'un point de terminaison d'API public accessible par Internet. Nous vous recommandons de supprimer le point de terminaison après un test.

```
cdk destroy
```

Utilisation d'ADOT pour instrumenter vos fonctions python

ADOT fournit des couches Lambda entièrement gérées qui regroupent tout ce dont vous avez besoin pour collecter des données de télémétrie à l'aide du kit SDK OTel. En consommant cette couche, vous pouvez instrumenter vos fonctions Lambda sans avoir à modifier le code de fonction. Vous pouvez également configurer votre couche pour effectuer une initialisation personnalisée d'OTel. Pour de plus amples informations, veuillez consulter Configuration personnalisée pour ADOT Collector sur Lambda dans la documentation ADOT.

Pour les exécutions python, vous pouvez ajouter leAWS couche Lambda gérée pour ADOT Pythonpour instrumenter automatiquement vos fonctions. Cette couche fonctionne à la fois pour les architectures arm64 et x86_64. Pour obtenir des instructions détaillées sur la façon d'ajouter cette couche, veuillez consulter AWS Distro pour prise en charge d'OpenTelemetry Lambda pour Python dans la documentation ADOT.

Utilisation du kit SDK X-Ray pour instrumenter vos fonctions Python

Pour enregistrer des détails sur les appels effectués par votre fonction Lambda à d'autres ressources de votre application, vous pouvez également utiliser le Kit SDK AWS X-Ray pour Python. Pour obtenir ce kit SDK, ajoutez le package `aws-xray-sdk` aux dépendances de votre application.

Example requirements.txt

```
jsonpickle==1.3
aws-xray-sdk==2.4.3
```

Dans votre code de fonction, vous pouvez instrumenter le kit SDK AWS clients en corrigeant le `boto3` bibliothèque avec le module `aws_xray_sdk.core`.

Example fonction – Tracer un kit SDK AWS client

```
import boto3
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch_all
```

```
logger = logging.getLogger()
logger.setLevel(logging.INFO)
patch_all()

client = boto3.client('lambda')
client.get_account_settings()

def lambda_handler(event, context):
    logger.info('## ENVIRONMENT VARIABLES\r' + jsonpickle.encode(dict(**os.environ)))
  ...
```

Une fois que vous avez ajouté les bonnes dépendances et effectué les modifications de code nécessaires, activez le suivi dans la configuration de votre fonction via la console Lambda ou l'API.

Activation du suivi avec la console Lambda

Pour activer/désactiver le traçage actif sur votre fonction Lambda avec la console, procédez comme suit :

Pour activer le traçage actif

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Choisissez Configuration, puis choisissez Outils de surveillance et d'opérations.
4. Dans Outils de surveillance supplémentaires, choisissez Modifier.
5. Sous CloudWatch Application Signals et AWS X-Ray sélectionnez Activer pour Suivis de service Lambda.
6. Choisissez Enregistrer.

Activation du suivi avec l'API Lambda

Configurez le suivi sur votre fonction Lambda avec le AWS CLI ou le kit SDK AWS, utilisez les opérations d'API suivantes :

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

L'exemple de commande AWS CLI suivant active le suivi actif sur une fonction nommée my-function.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Le mode de suivi fait partie de la configuration spécifique de la version lorsque vous publiez une version de votre fonction. Vous ne pouvez pas modifier le mode de suivi sur une version publiée.

Activation du suivi avec CloudFormation

Pour activer le suivi d'une ressource `AWS::Lambda::Function` dans un modèle CloudFormation, utilisez la propriété `TracingConfig`.

Example function-inline.yml – Configuration du suivi

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

Pour une ressource AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function`, utilisez la propriété `Tracing`.

Example template.yml – Configuration du suivi

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

Interprétation d'un suivi X-Ray

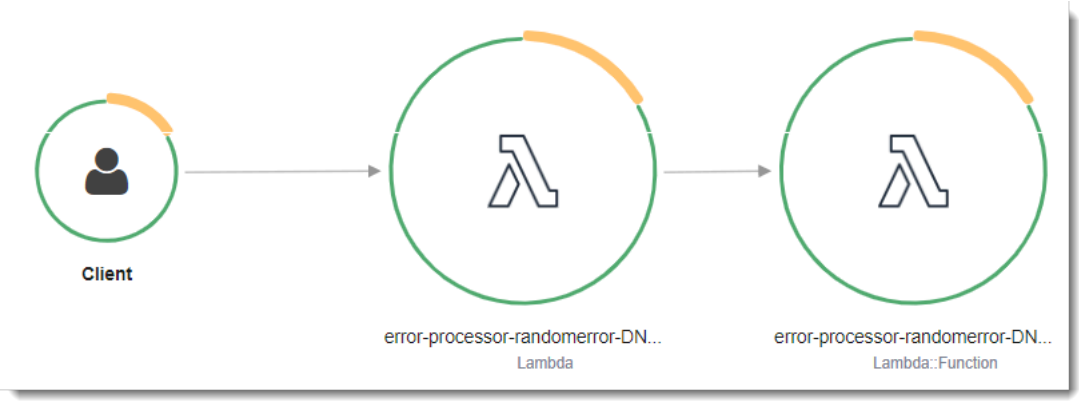
Votre fonction a besoin d'une autorisation pour charger des données de suivi vers X-Ray. Lorsque vous activez le suivi actif dans la console Lambda, Lambda ajoute les autorisations requises au rôle

d'exécution de votre fonction. Sinon, ajoutez la politique AWSXRayDaemonWriteAccess au rôle d'exécution.

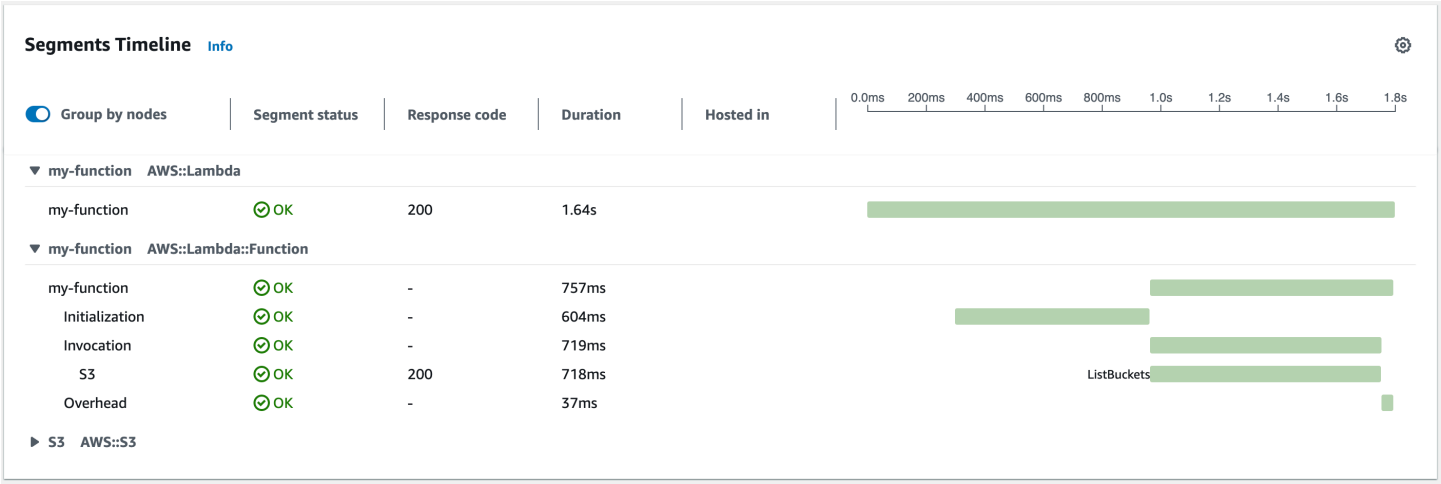
Une fois que vous avez configuré le suivi actif, vous pouvez observer des demandes spécifiques via votre application. Le graphique de services X-Ray affiche des informations sur votre application et tous ses composants. L'exemple suivant montre une application dotée de deux fonctions. La fonction principale traite les événements et renvoie parfois des erreurs. La deuxième fonction en partant du haut traite les erreurs qui apparaissent dans le groupe de journaux du premier et utilise le kit SDK AWS pour appeler X-Ray, Amazon Simple Storage Service (Amazon S3) et Amazon CloudWatch logs.

X-Ray ne trace pas toutes les requêtes vers votre application. X-Ray applique un algorithme d'échantillonnage pour s'assurer que le suivi est efficace, tout en fournissant un échantillon représentatif de toutes les demandes. Le taux d'échantillonnage est 1 demande par seconde et 5 % de demandes supplémentaires. Vous ne pouvez pas configurer ce taux d'échantillonnage X-Ray pour vos fonctions.

Dans X-Ray, un suivi enregistre des informations sur une demande traitée par un ou plusieurs services. Lambda enregistre deux segments par suivi, ce qui a pour effet de créer deux nœuds sur le graphique du service. L'image suivante met en évidence ces deux nœuds :



Le premier nœud sur la gauche représente le service Lambda qui reçoit la demande d'invocation. Le deuxième nœud représente votre fonction Lambda spécifique. L'exemple suivant illustre une trace avec ces deux segments. Les deux sont nommés my-function, mais l'un a pour origine `AWS::Lambda` et l'autre a pour origine `AWS::Lambda::Function`. Si le segment `AWS::Lambda` affiche une erreur, cela signifie que le service Lambda a rencontré un problème. Si le segment `AWS::Lambda::Function` affiche une erreur, cela signifie que votre fonction a rencontré un problème.



Cet exemple développe le segment `AWS::Lambda::Function` pour afficher ses trois sous-segments.

> **Note**
>
> AWS est en train de modifier le service Lambda. En raison de ces modifications, vous pouvez constater des différences mineures entre la structure et le contenu des messages du journal système et des segments de suivi émis par les différentes fonctions Lambda de votre Compte AWS.

L'exemple de suivi présenté ici illustre le segment de fonction à l'ancienne. Les différences entre les segments à l'ancienne et de style moderne sont décrites dans les paragraphes suivants.
Ces modifications seront mises en œuvre au cours des prochaines semaines, et toutes les fonctions dans toutes les Régions AWS, sauf les régions de Chine et GovCloud, passeront à l'utilisation du nouveau format des messages de journal et des segments de suivi.

Le segment de fonction à l'ancienne contient les sous-segments suivants :

- Initialization (Initialisation) : représente le temps passé à charger votre fonction et à exécuter le code d'initialisation. Ce sous-segment apparaît pour le premier événement traité par chaque instance de votre fonction.
- Invocation – Représente le temps passé à exécuter votre code de gestionnaire.
- Overhead (Travail supplémentaire) – Représente le temps que le fichier d'exécution Lambda passe à se préparer à gérer l'événement suivant.

Le segment de fonction de style moderne ne contient pas de sous-segment `Invocation`. À la place, les sous-segments du client sont directement rattachés au segment de fonction. Pour plus d'informations sur la structure des segments de fonction à l'ancienne et de style moderne, consultez the section called "Comprendre les suivis X-Ray".

Vous pouvez également utiliser des clients HTTP, enregistrer des requêtes SQL et créer des sous-segments personnalisés avec des annotations et des métadonnées. Pour plus d'informations, consultez Kit SDK AWS X-Ray pour Python dans le AWS X-Ray Guide du développeur.

Tarification

Vous pouvez utiliser gratuitement le traçage par X-Ray chaque mois jusqu'à une certaine limite dans le cadre de l'offre gratuite AWS. Au-delà de ce seuil, X-Ray facture le stockage et la récupération du suivi. Pour plus d'informations, consultez Tarification d'AWS X-Ray.

Stockage des dépendances d'exécution dans une couche (kit SDK X-Ray)

Si vous utilisez le kit SDK X-Ray pour instrumenter des clients de kit SDK AWS avec votre code de fonction, votre package de déploiement peut devenir assez volumineux. Pour éviter de charger

des dépendances d'environnement d'exécution chaque fois que vous mettez à jour votre code de fonction, empaquetez le kit SDK X-Ray dans une couche Lambda.

L'exemple suivant montre une ressource `AWS::Serverless::LayerVersion` qui stocke le Kit SDK AWS X-Ray pour Python.

Example template.yml : couche de dépendances

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: function/.
      Tracing: Active
      Layers:
        - !Ref libs
      ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-python-lib
      Description: Dependencies for the blank-python sample app.
      ContentUri: package/.
      CompatibleRuntimes:
        - python3.11
```

Avec cette configuration, vous ne mettez à jour les fichiers de couche de bibliothèque que si vous modifiez vos dépendances d'exécution. Étant donné que le package de déploiement de la fonction contient uniquement votre code, cela peut contribuer à réduire les temps de chargement.

La création d'une couche de dépendances nécessite des modifications de génération pour créer l'archive des couches avant le déploiement. Pour un exemple fonctionnel, consultez l'exemple d'application blank-python.

Création de fonctions Lambda avec Ruby

Vous pouvez exécuter du code Ruby dans AWS Lambda. Lambda fournit des runtimes pour Ruby qui exécutent votre code afin de traiter des événements. Votre code s'exécute dans un environnement qui inclut le AWS SDK pour Ruby, avec les informations d'identification d'un rôle Gestion des identités et des accès AWS (IAM) que vous gérez. Pour en savoir plus sur les versions du kit SDK incluses dans les environnements d'exécution Ruby, consultez the section called "Versions du SDK incluses dans l'environnement d'exécution".

Lambda prend en charge les runtimes Ruby suivants.

Name	Identifiant	Système d'exploitation	Date d'obsolescence	Créer la fonction de blocage	Mettre à jour la fonction de blocage
Ruby 3.4	`ruby3.4`	Amazon Linux 2	31 mars 2028	30 avril 2028	31 mai 2028
Ruby 3.3	`ruby3.3`	Amazon Linux 2	31 mars 2027	30 avril 2027	31 mai 2027
Ruby 3.2	`ruby3.2`	Amazon Linux 2	31 mars 2026	31 août 2026	30 sept. 2026

Pour créer une fonction Ruby

1. Ouvrez la console Lambda.
2. Sélectionnez Créer une fonction.
3. Configurez les paramètres suivants :
   - Nom de la fonction : saisissez le nom de la fonction.
   - Runtime : Choisissez Ruby 3.4.
4. Choisissez Créer une fonction.

La console crée une fonction Lambda avec un seul fichier source nommé `lambda_function.rb`. Vous pouvez modifier ce fichier et ajouter d'autres fichiers dans l'éditeur de code intégré. Dans la section DÉPLOYER, choisissez Déployer pour mettre à jour le code de votre fonction. Ensuite, pour

exécuter votre code, choisissez Créer un événement de test dans la section ÉVÉNEMENTS DE TEST.

Le fichier `lambda_function.rb` exporte une fonction nommée `lambda_handler` qui accepte un objet événement et un objet contexte. Il s'agit de la fonction de gestionnaire que Lambda appelle lors de l'invocation de la fonction. L'environnement d'exécution (runtime) de la fonction Ruby obtient les événements d'invocations de Lambda et les transmet au gestionnaire. Dans la configuration de fonction, la valeur de gestionnaire est `lambda_function.lambda_handler`.

Lorsque vous enregistrez votre code de fonction, la console Lambda crée un package de déploiement d'archive de fichiers .zip. Lorsque vous développez votre code de fonction en dehors de la console (à l'aide d'un IDE), vous devez créer un package de déploiement pour charger votre code dans la fonction Lambda.

Le runtime de la fonction transmet un objet de contexte au gestionnaire, en plus de l'événement d'invocation. L'objet de contexte contient des informations supplémentaires sur l'invocation, la fonction et l'environnement d'exécution. Des informations supplémentaires sont disponibles dans les variables d'environnement.

Votre fonction Lambda est fournie avec un groupe de CloudWatch journaux Logs. La fonction runtime envoie les détails de chaque appel à CloudWatch Logs. Il relaie tous les journaux que votre fonction génère pendant l'invocation. Si votre fonction renvoie une erreur, Lambda met en forme l'erreur et la renvoie à l'appelant.

Rubriques

- Versions du SDK incluses dans l'environnement d'exécution
- Activer un autre JIT Ruby (YJIT)
- Définition du gestionnaire de fonction Lambda dans Ruby
- Déployer des fonctions Lambda en Ruby avec des archives de fichiers .zip
- Déploiement de fonctions Lambda en Ruby avec des images conteneurs
- Utilisation de couches pour les fonctions Lambda Ruby
- Utilisation de l'objet de contexte Lambda pour récupérer les informations de la fonction Ruby
- Journalisation et surveillance des fonctions Lambda Ruby
- Instrumentation du code Ruby dans AWS Lambda

Versions du SDK incluses dans l'environnement d'exécution

La version du AWS SDK incluse dans le runtime Ruby dépend de la version d'exécution et de votre Région AWS. Le AWS SDK pour Ruby est conçu pour être modulaire et est séparé par Service AWS. Pour trouver le numéro de version d'un gem de service particulier inclus dans l'environnement d'exécution que vous utilisez, créez une fonction Lambda avec un code au format suivant. Remplacez `aws-sdk-s3` et `Aws::S3` par le nom des gems de service utilisés par votre code.

```
require 'aws-sdk-s3'

def lambda_handler(event:, context:)
  puts "Service gem version: #{Aws::S3::GEM_VERSION}"
  puts "Core version: #{Aws::CORE_GEM_VERSION}"
end
```

Activer un autre JIT Ruby (YJIT)

L'exécution Ruby 3.2 prend en charge YJIT, un compilateur JIT Ruby léger et minimaliste. YJIT offre des performances nettement supérieures, mais utilise également plus de mémoire que l'interpréteur Ruby. YJIT est recommandé pour les charges de travail Ruby on Rails.

YJIT n'est pas activé par défaut. Pour activer YJIT pour une fonction Ruby 3.2, définissez la variable d'environnement `RUBY_YJIT_ENABLE` sur 1. Pour confirmer que YJIT est activé, imprimez le résultat de la méthode `RubyVM::YJIT.enabled?`.

Example – Confirmez que YJIT est activé

```
puts(RubyVM::YJIT.enabled?())
# => true
```

Définition du gestionnaire de fonction Lambda dans Ruby

Le gestionnaire de fonction Lambda est la méthode dans votre code de fonction qui traite les événements. Lorsque votre fonction est invoquée, Lambda exécute la méthode du gestionnaire. Votre fonction s'exécute jusqu'à ce que le gestionnaire renvoie une réponse, se ferme ou expire.

Rubriques

- Notions de base sur le gestionnaire Ruby
- Pratiques exemplaires en matière de code pour les fonctions Lambda Ruby

Notions de base sur le gestionnaire Ruby

Dans l'exemple suivant, le fichier `function.rb` définit une méthode de gestionnaire nommée `handler`. La fonction de gestionnaire prend deux objets en tant qu'entrées et renvoie un document JSON.

Example function.rb

```
require 'json'

def handler(event:, context:)
    { event: JSON.generate(event), context: JSON.generate(context.inspect) }
end
```

Dans la configuration de votre fonction, le paramètre `handler` indique à Lambda où trouver le gestionnaire. Pour l'exemple précédent, la valeur correcte pour ce paramètre est **`function.handler`**. Elle inclut deux noms séparées par un point : le nom du fichier et le nom de la méthode de gestionnaire.

Vous pouvez également définir votre méthode de gestionnaire dans une classe. L'exemple suivant définit une méthode de gestionnaire nommée `process` sur une classe nommée `Handler` dans un module appelé `LambdaFunctions`.

Example source.rb

```
module LambdaFunctions
  class Handler
    def self.process(event:,context:)
```

```
      "Hello!"
    end
  end
end
```

Dans ce cas, le paramètre de gestionnaire est **`source.LambdaFunctions::Handler.process`**.

Les deux objets qui sont acceptés par le gestionnaire sont l'événement d'invocation et le contexte. L'événement est un objet Ruby qui contient la charge utile qui est fournie par le mécanisme d'invocation. Si la charge utile est un document JSON, l'objet d'événement est un hachage Ruby. Sinon, il s'agit d'une chaîne. L'objet de contexte contient des méthodes et des propriétés qui fournissent des informations sur l'invocation, la fonction et l'environnement d'exécution.

Le gestionnaire de fonction est exécuté chaque fois que votre fonction Lambda est invoquée. Le code statique à l'extérieur du gestionnaire est exécuté une fois par instance de la fonction. Si votre gestionnaire utilise des ressources telles que les clients de kits SDK et les connexions de base de données, vous pouvez les créer à l'extérieur de la méthode de gestionnaire afin de les réutiliser pour plusieurs invocations.

Chaque instance de votre fonction peut traiter plusieurs événements d'invocations, mais elle ne traite qu'un événement à la fois. Le nombre d'instances traitant un événement à un moment donné est la simultanéité de votre fonction. Pour plus d'informations sur l'environnement d'exécution Lambda, consultez Comprendre le cycle de vie de l'environnement d'exécution Lambda.

Pratiques exemplaires en matière de code pour les fonctions Lambda Ruby

Respectez les directives de la liste suivante pour utiliser les pratiques exemplaires de codage lors de la création de vos fonctions Lambda :

- Séparez le gestionnaire Lambda de votre logique principale. Cela vous permet de créer une fonction testable plus unitaire. Par exemple, en Ruby, vous pouvez observer ce qui suit :

```
def lambda_handler(event:, context:)
    foo = event['foo']
    bar = event['bar']

    result = my_lambda_function(foo:, bar:)

end
```

```
def my_lambda_function(foo:, bar:)
    // MyLambdaFunction logic here

end
```

- Contrôlez les dépendances du package de déploiement de vos fonctions. L'environnement d'exécution AWS Lambda contient un certain nombre de bibliothèques. Pour l'environnement d'exécution Ruby, celles-ci incluent notamment le kit SDK AWS. Pour activer le dernier ensemble de mises à jour des fonctionnalités et de la sécurité, Lambda met régulièrement à jour ces bibliothèques. Ces mises à jour peuvent introduire de subtiles modifications dans le comportement de votre fonction Lambda. Pour disposer du contrôle total des dépendances que votre fonction utilise, empaquetez toutes vos dépendances avec votre package de déploiement.
- Réduisez la complexité de vos dépendances. Privilégiez les infrastructures plus simples qui se chargent rapidement au démarrage de l'environnement d'exécution.
- Réduisez la taille de votre package de déploiement selon ses besoins relatifs à l'environnement d'exécution. Cela contribue à réduire le temps nécessaire au téléchargement et à la décompression de votre package de déploiement avant l'invocation. Pour les fonctions créées en Ruby, évitez de charger la totalité de la bibliothèque du kit SDK AWS comme partie intégrante de votre package de déploiement. À la place, appuyez-vous de façon sélective sur les gems qui sélectionnent les composants du kit SDK dont vous avez besoin (par exemple, les gems du kit SDK DynamoDB ou SDK Amazon S3).

Tirez parti de la réutilisation de l'environnement d'exécution pour améliorer les performances de votre fonction. Initialisez les clients SDK et les connexions à la base de données en dehors du gestionnaire de fonctions et mettez en cache les actifs statiques localement dans le répertoire `/tmp`. Les invocations ultérieures traitées par la même instance de votre fonction peuvent réutiliser ces ressources. Cela permet d'économiser des coûts, tout en réduisant le temps d'exécution de la fonction.

Pour éviter des éventuelles fuites de données entre les invocations, n'utilisez pas l'environnement d'exécution pour stocker des données utilisateur, des événements ou d'autres informations ayant un impact sur la sécurité. Si votre fonction repose sur un état réversible qui ne peut pas être stocké en mémoire dans le gestionnaire, envisagez de créer une fonction distincte ou des versions distinctes d'une fonction pour chaque utilisateur.

Utilisez une directive keep-alive pour maintenir les connexions persistantes. Lambda purge les connexions inactives au fil du temps. Si vous tentez de réutiliser une connexion inactive lorsque vous invoquez une fonction, cela entraîne une erreur de connexion. Pour maintenir votre connexion

persistante, utilisez la directive Keep-alive associée à votre environnement d'exécution. Pour obtenir un exemple, consultez Réutilisation des connexions avec Keep-Alive dans Node.js.

Utilisez des variables d'environnement pour transmettre des paramètres opérationnels à votre fonction. Par exemple, si vous écrivez dans un compartiment Amazon S3 au lieu de coder en dur le nom du compartiment dans lequel vous écrivez, configurez le nom du compartiment comme variable d'environnement.

Évitez d'utiliser des invocations récursives dans votre fonction Lambda, lorsque la fonction s'invoque elle-même ou démarre un processus susceptible de l'invoquer à nouveau. Cela peut entraîner un volume involontaire d'invocations de fonction et des coûts accrus. Si vous constatez un volume involontaire d'invocations, définissez immédiatement la simultanéité réservée à la fonction sur `0` afin de limiter toutes les invocations de la fonction, pendant que vous mettez à jour le code.

N'utilisez pas d'API non publiques non documentées dans le code de votre fonction Lambda. Pour les exécutions gérées AWS Lambda, Lambda applique périodiquement des mises à jour de sécurité et fonctionnelles aux API internes de Lambda. Ces mises à jour internes de l'API peuvent être incompatibles avec les versions antérieures, entraînant des conséquences imprévues telles que des échecs d'invocation si votre fonction dépend de ces API non publiques. Consultez la Référence d'API pour obtenir la liste des API accessibles au public.

Écriture du code idempotent. L'écriture de code idempotent pour vos fonctions garantit ne gestion identique des événements dupliqués. Votre code doit valider correctement les événements et gérer correctement les événements dupliqués. Pour de plus amples informations, veuillez consulterComment faire en sorte que ma fonction Lambda soit idempotente ?.

Déployer des fonctions Lambda en Ruby avec des archives de fichiers .zip

Le code de votre fonction AWS Lambda comprend un fichier .rb contenant le code du gestionnaire de votre fonction, ainsi que toutes les dépendances (GEM) supplémentaires dont votre code dépend. Pour déployer ce code de fonction vers Lambda, vous utilisez un package de déploiement. Ce package peut être une archive de fichier .zip ou une image de conteneur. Pour plus d'informations sur l'utilisation d'images de conteneur avec Ruby, consultez la page Déployer des fonctions Lambda Ruby avec des images de conteneur.

Pour créer votre package de déploiement sous forme d'archive de fichier .zip, vous pouvez utiliser l'utilitaire d'archivage .zip intégré à votre outil de ligne de commande, ou tout autre utilitaire .zip tel que 7zip. Les exemples présentés dans les sections suivantes supposent que vous utilisez un outil `zip` de ligne de commande dans un environnement Linux ou macOS. Pour utiliser les mêmes commandes sous Windows, vous pouvez installer le sous-système Windows pour Linux afin d'obtenir une version intégrée à Windows d'Ubuntu et de Bash.

Notez que Lambda utilise les autorisations de fichiers POSIX. Ainsi, vous pourriez devoir définir des autorisations pour le dossier du package de déploiement avant de créer l'archive de fichiers .zip.

Les exemples de commandes présentés dans les sections suivantes utilisent l'utilitaire Bundler pour ajouter des dépendances à votre package de déploiement. Pour installer Bundler, exécutez la commande suivante.

```
gem install bundler
```

Sections

- Dépendances dans Ruby
- Création d'un package de déploiement .zip sans dépendances
- Création d'un package de déploiement .zip avec dépendances
- Création d'une couche Ruby pour vos dépendances
- Création de packages de déploiement .zip avec des bibliothèques natives
- Création et mise à jour de fonctions Lambda Ruby à l'aide de fichiers .zip

Dépendances dans Ruby

Pour les fonctions Lambda qui utilisent l'exécution Ruby, une dépendance peut être n'importe quelle GEM Ruby. Lorsque vous déployez votre fonction à l'aide d'une archive .zip, vous pouvez soit ajouter ces dépendances à votre fichier .zip avec votre code de fonction, soit utiliser une couche Lambda. Une couche est un fichier .zip séparé qui peut contenir du code supplémentaire et d'autres contenus. Pour en savoir plus sur l'utilisation des couches Lambda, consultez Couches Lambda.

L'environnement d'exécution Ruby inclut l'AWS SDK pour Ruby. Si votre fonction utilise le kit SDK, vous n'avez pas besoin de l'intégrer à votre code. Toutefois, pour garder le contrôle total de vos dépendances ou pour utiliser une version spécifique du kit SDK, vous pouvez l'ajouter au package de déploiement de votre fonction. Vous pouvez inclure le kit SDK dans votre fichier .zip ou bien l'ajouter à l'aide d'une couche Lambda. Les dépendances de votre fichier .zip ou des couches Lambda sont prioritaires sur les versions incluses dans l'exécution. Pour savoir quelle version du kit SDK pour Ruby est incluse dans votre version d'exécution, consultez the section called "Versions du SDK incluses dans l'environnement d'exécution".

Dans le cadre du modèle de responsabilité partagée AWS, vous êtes responsable de la gestion de toutes les dépendances dans les packages de déploiement de vos fonctions. Cela inclut l'application de mises à jour et de correctifs de sécurité. Pour mettre à jour les dépendances dans le package de déploiement de votre fonction, créez d'abord un nouveau fichier .zip, puis chargez-le sur Lambda. Pour plus d'informations, consultez Création d'un package de déploiement .zip avec dépendances et Création et mise à jour de fonctions Lambda Ruby à l'aide de fichiers .zip.

Création d'un package de déploiement .zip sans dépendances

Si le code de votre fonction ne comporte aucune dépendance, votre fichier .zip contient uniquement le fichier .rb contenant le code du gestionnaire de votre fonction. Utilisez votre utilitaire zip préféré pour créer un fichier .zip avec votre fichier .rb à la racine. Si le fichier .rb ne se trouve pas à la racine de votre fichier .zip, Lambda ne sera pas en mesure d'exécuter votre code.

Pour savoir comment déployer votre fichier .zip pour créer une nouvelle fonction Lambda ou mettre à jour une fonction Lambda existante, veuillez consulter la rubrique Création et mise à jour de fonctions Lambda Ruby à l'aide de fichiers .zip.

Création d'un package de déploiement .zip avec dépendances

Si votre code de fonction dépend de GEM Ruby supplémentaires, vous pouvez soit ajouter ces dépendances à votre fichier .zip avec votre code de fonction, soit utiliser une couche Lambda. Les

instructions de cette section vous indiquent comment inclure les dépendances dans votre package de déploiement .zip. Pour obtenir des instructions sur la façon d'inclure vos dépendances dans une couche, voir the section called "Création d'une couche Ruby pour vos dépendances".

Supposons que votre code de fonction soit enregistré dans un fichier nommé `lambda_function.rb` dans le répertoire de votre projet. L'exemple de commandes d'interface de ligne de commande suivant crée un fichier .zip nommé `my_deployment_package.zip` contenant le code de votre fonction et ses dépendances.

Pour créer le package de déploiement

1. Dans le répertoire de votre projet, créez un fichier `Gemfile` pour y spécifier vos dépendances.

   ```
   bundle init
   ```

2. En utilisant l'éditeur de texte de votre choix, modifiez le fichier `Gemfile` pour spécifier les dépendances de votre fonction. Par exemple, pour utiliser la GEM TZInfo, modifiez votre fichier `Gemfile` pour qu'il ressemble à ce qui suit.

   ```
   source "https://rubygems.org"
   gem "tzinfo"
   ```

3. Exécutez la commande suivante pour installer les GEM spécifiées dans votre fichier `Gemfile` dans votre répertoire de projet. Cette commande définit `vendor/bundle` comme chemin par défaut pour les installations de GEM.

   ```
   bundle config set --local path 'vendor/bundle' && bundle install
   ```

   Vous devez visualiser des résultats similaires à ce qui suit.

   ```
   Fetching gem metadata from https://rubygems.org/...........
   Resolving dependencies...
   Using bundler 2.4.13
   Fetching tzinfo 2.0.6
   Installing tzinfo 2.0.6
   ...
   ```

   > **Note**
   >
   > Pour réinstaller globalement les GEM ultérieurement, exécutez la commande suivante.

```
bundle config set --local system 'true'
```

4. Créez une archive de fichiers .zip contenant le fichier `lambda_function.rb` avec le code du gestionnaire de votre fonction et les dépendances que vous avez installées à l'étape précédente.

```
zip -r my_deployment_package.zip lambda_function.rb vendor
```

Vous devez visualiser des résultats similaires à ce qui suit.

```
adding: lambda_function.rb (deflated 37%)
  adding: vendor/ (stored 0%)
  adding: vendor/bundle/ (stored 0%)
  adding: vendor/bundle/ruby/ (stored 0%)
  adding: vendor/bundle/ruby/3.2.0/ (stored 0%)
  adding: vendor/bundle/ruby/3.2.0/build_info/ (stored 0%)
  adding: vendor/bundle/ruby/3.2.0/cache/ (stored 0%)
  adding: vendor/bundle/ruby/3.2.0/cache/aws-eventstream-1.0.1.gem (deflated 36%)
...
```

Création d'une couche Ruby pour vos dépendances

Pour savoir comment empaqueter vos dépendances Ruby dans une couche Lambda, consultez the section called "Couches".

Création de packages de déploiement .zip avec des bibliothèques natives

De nombreuses GEM Ruby courantes telles que `nokogiri`, `nio4r` et `mysql` contiennent des extensions natives écrites en C. Lorsque vous ajoutez des bibliothèques contenant du code C à votre package de déploiement, vous devez créer votre package correctement pour vous assurer qu'il est compatible avec l'environnement d'exécution Lambda.

Pour les applications de production, nous vous recommandons de créer et de déployer votre code en utilisant AWS Serverless Application Model (AWS SAM). Dans AWS SAM utilisez l'option `sam build --use-container` pour créer votre fonction dans un conteneur Docker de type Lambda. Pour en savoir plus sur l'utilisation de AWS SAM pour déployer le code de vos fonctions, consultez la section Création d'applications dans le Guide du développeur AWS SAM.

Pour créer un package de déploiement .zip contenant des GEM avec des extensions natives sans utiliser AWS SAM, vous pouvez également utiliser un conteneur pour regrouper vos dépendances dans un environnement identique à l'environnement d'exécution Lambda Ruby. Pour réaliser ces étapes, Docker doit être installé sur votre machine de génération. Pour en savoir plus sur l'installation de Docker, consultez Installation du moteur Docker.

Pour créer un package de déploiement .zip dans un conteneur Docker

1. Créez un dossier sur votre machine de génération locale pour y enregistrer votre conteneur. Dans ce dossier, créez un fichier nommé `dockerfile` et collez-y le code suivant.

```
FROM public.ecr.aws/sam/build-ruby3.2:latest-x86_64
RUN gem update bundler
CMD "/bin/bash"
```

2. Dans le dossier dans lequel vous avez créé votre fichier `dockerfile`, exécutez la commande suivante pour créer le conteneur Docker.

```
docker build -t awsruby32 .
```

3. Accédez au répertoire du projet contenant le fichier `.rb` avec le code du gestionnaire de votre fonction et le fichier `Gemfile` spécifiant les dépendances de votre fonction. À partir de ce répertoire, exécutez la commande suivante pour démarrer le conteneur Lambda Ruby.

   Linux/MacOS

```
docker run --rm -it -v $PWD:/var/task -w /var/task awsruby32
```

> **Note**
>
> Sous macOS, il se peut qu'un avertissement vous informe que la plateforme de l'image demandée ne correspond pas à la plateforme de l'hôte détectée. Ignorez cet avertissement.

   Windows PowerShell

```
docker run --rm -it -v ${pwd}:var/task -w /var/task awsruby32
```

Lorsque votre conteneur démarre, vous devriez voir une invite bash.

```
bash-4.2#
```

4. Configurez l'utilitaire bundle pour installer les GEM spécifiées dans votre fichier `Gemfile` dans votre répertoire `vendor/bundle` local et installer vos dépendances.

```
bash-4.2# bundle config set --local path 'vendor/bundle' && bundle install
```

5. Créez le package de déploiement .zip avec le code de votre fonction et ses dépendances. Dans cet exemple, le fichier contenant le code du gestionnaire de votre fonction est nommé `lambda_function.rb`.

```
bash-4.2# zip -r my_deployment_package.zip lambda_function.rb vendor
```

6. Quittez le conteneur et retournez dans votre répertoire de projet local.

```
bash-4.2# exit
```

Vous pouvez désormais utiliser le package de déploiement du fichier .zip pour créer ou mettre à jour votre fonction Lambda. Consultez Création et mise à jour de fonctions Lambda Ruby à l'aide de fichiers .zip

Création et mise à jour de fonctions Lambda Ruby à l'aide de fichiers .zip

Une fois que vous avez créé votre package de déploiement .zip, vous pouvez l'utiliser pour créer une nouvelle fonction Lambda ou mettre à jour une fonction Lambda existante. Vous pouvez déployer votre package .zip à l'aide de la console Lambda, de l'AWS Command Line Interface, et de l'API Lambda. Vous pouvez également créer et mettre à jour des fonctions Lambda à l'aide de l'AWS Serverless Application Model (AWS SAM) et de CloudFormation.

La taille maximale d'un package de déploiement .zip pour Lambda est de 250 Mo (décompressé). Notez que cette limite s'applique à la taille combinée de tous les fichiers que vous chargez, y compris les couches Lambda.

Le runtime Lambda a besoin d'une autorisation pour lire les fichiers de votre package de déploiement. Dans la notation octale des autorisations Linux, Lambda a besoin de 644 autorisations

pour les fichiers non exécutables (rw-r--r--) et de 755 autorisations (rwxr-xr-x) pour les répertoires et les fichiers exécutables.

Sous Linux et macOS, utilisez la commande chmod pour modifier les autorisations de fichiers sur les fichiers et les répertoires de votre package de déploiement. Par exemple, pour octroyer à un fichier non exécutable les autorisations correctes, exécutez la commande suivante.

```
chmod 644 <filepath>
```

Pour modifier les autorisations relatives aux fichiers dans Windows, voir Définir, afficher, modifier ou supprimer des autorisations sur un objet dans la documentation Microsoft Windows.

> **Note**
>
> Si vous n'octroyez pas à Lambda les autorisations dont il a besoin pour accéder aux répertoires de votre package de déploiement, Lambda définit les autorisations de ces répertoires à 755 (rwxr-xr-x).

Création et mise à jour de fonctions avec des fichiers .zip à l'aide de la console

Pour créer une nouvelle fonction, vous devez d'abord créer la fonction dans la console, puis charger votre archive .zip. Pour mettre à jour une fonction existante, ouvrez la page de votre fonction, puis suivez la même procédure pour ajouter votre fichier .zip mis à jour.

Si votre fichier .zip fait moins de 50 Mo, vous pouvez créer ou mettre à jour une fonction en chargeant le fichier directement à partir de votre ordinateur local. Pour les fichiers .zip de plus de 50 Mo, vous devez d'abord charger votre package dans un compartiment Amazon S3. Pour savoir comment charger un fichier dans un compartiment Amazon S3 à l'aide de la AWS Management Console, veuillez consulter la rubrique Premiers pas avec Amazon S3. Pour charger des fichiers à l'aide de l'AWS CLI, veuillez consulter la rubrique Déplacer des objets dans le Guide de l'utilisateur AWS CLI.

> **Note**
>
> Vous ne pouvez pas modifier le type de package de déploiement (.zip ou image de conteneur) d'une fonction existante. Par exemple, vous ne pouvez pas convertir une fonction d'image de conteneur pour utiliser un fichier d'archive .zip à la place. Vous devez créer une nouvelle fonction.

Pour créer une nouvelle fonction (console)

1. Ouvrez la page Fonctions de la console Lambda et choisissez Créer une fonction.
2. Choisissez Créer à partir de zéro.
3. Sous Informations de base, procédez comme suit :
    a. Pour Nom de la fonction, saisissez le nom de la fonction.
    b. Pour Exécution, sélectionnez l'exécution que vous souhaitez utiliser.
    c. (Facultatif) Pour Architecture, choisissez l'architecture de l'ensemble des instructions pour votre fonction. L'architecture par défaut est x86_64. Assurez-vous que le package de déploiement .zip pour votre fonction est compatible avec l'architecture de l'ensemble d'instructions que vous sélectionnez.
4. (Facultatif) Sous Permissions (Autorisations), développez Change default execution role (Modifier le rôle d'exécution par défaut). Vous pouvez créer un rôle d'exécution ou en utiliser un existant.
5. Choisissez Créer une fonction. Lambda crée une fonction de base « Hello world » à l'aide de l'exécution de votre choix.

Pour charger une archive .zip à partir de votre ordinateur local (console)

1. Sur la page Fonctions de la console Lambda, choisissez la fonction pour laquelle vous souhaitez charger le fichier .zip.
2. Sélectionnez l'onglet Code.
3. Dans le volet Source du code, choisissez Charger à partir de.
4. Choisissez Fichier .zip.
5. Pour charger un fichier .zip, procédez comme suit :
    a. Sélectionnez Charger, puis choisissez votre fichier .zip dans le sélecteur de fichiers.
    b. Choisissez Ouvrir.
    c. Choisissez Enregistrer.

Pour charger une archive .zip depuis un compartiment Amazon S3 (console)

1. Sur la page Fonctions de la console Lambda, choisissez la fonction pour laquelle vous souhaitez charger un nouveau fichier .zip.

2. Sélectionnez l'onglet Code.
3. Dans le volet Source du code, choisissez Charger à partir de.
4. Choisissez l'emplacement Amazon S3.
5. Collez l'URL du lien Amazon S3 de votre fichier .zip et choisissez Enregistrer.

Mise à jour des fonctions du fichier .zip à l'aide de l'éditeur de code de la console

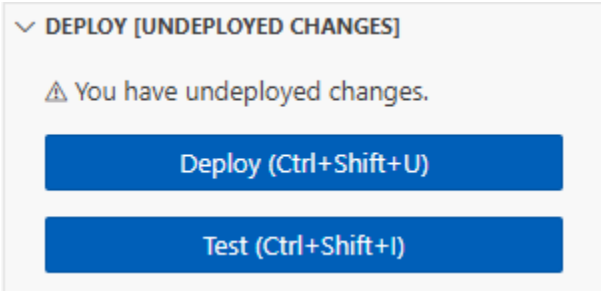
Pour certaines fonctions avec des packages de déploiement .zip, vous pouvez utiliser l'éditeur de code intégré de la console Lambda pour mettre à jour le code de votre fonction directement. Pour utiliser cette fonctionnalité, votre fonction doit répondre aux critères suivants :

- Votre fonction doit utiliser l'une des exécutions des langages interprétés (Python, Node.js ou Ruby).
- La taille du package de déploiement de votre fonction doit être inférieure à 50 Mo (décompressé).

Le code des fonctions avec les packages de déploiement d'images de conteneurs ne peut pas être édité directement dans la console.

Pour mettre à jour le code de fonction à l'aide de l'éditeur de code de la console

1. Ouvrez la page Fonctions de la console Lambda et choisissez votre fonction.
2. Sélectionnez l'onglet Code.
3. Dans le volet Source du code, sélectionnez votre fichier de code source et modifiez-le dans l'éditeur de code intégré.
4. Dans la section DÉPLOYER, choisissez Déployer pour mettre à jour le code de votre fonction :



Création et mise à jour de fonctions avec des fichiers .zip à l'aide de l'AWS CLI

Vous pouvez utiliser l'AWS CLI pour créer une nouvelle fonction ou pour mettre à jour une fonction existante à l'aide d'un fichier .zip. Utilisez les commandes create-function et update-function-code pour déployer votre package .zip. Si votre fichier .zip est inférieur à 50 Mo, vous pouvez charger le package .zip à partir d'un emplacement de fichier sur votre machine de génération locale. Pour les fichiers plus volumineux, vous devez charger votre package .zip à partir d'un compartiment Amazon S3. Pour savoir comment charger un fichier dans un compartiment Amazon S3 à l'aide de l'AWS CLI, veuillez consulter la rubrique Déplacer des objets dans le Guide de l'utilisateur AWS CLI.

> **Note**
>
> Si vous chargez votre fichier .zip à partir d'un compartiment Amazon S3 à l'aide de l'AWS CLI, le compartiment doit se trouver dans la même Région AWS que votre fonction.

Pour créer une nouvelle fonction à l'aide d'un fichier .zip avec l'AWS CLI, vous devez spécifier les éléments suivants :

- Le nom de votre fonction (`--function-name`)
- L'exécution de votre fonction (`--runtime`)
- L'Amazon Resource Name (ARN) du rôle d'exécution de votre fonction (`--role`)
- Le nom de la méthode du gestionnaire dans votre code de fonction (`--handler`)

Vous devez également indiquer l'emplacement de votre fichier .zip. Si votre fichier .zip se trouve dans un dossier sur votre machine de génération locale, utilisez l'option `--zip-file` pour spécifier le chemin d'accès du fichier, comme le montre l'exemple de commande suivant.

```
aws lambda create-function --function-name myFunction \
--runtime ruby3.2 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

Pour spécifier l'emplacement du fichier .zip dans un compartiment Amazon S3, utilisez l'option `--code` comme le montre l'exemple de commande suivant. Vous devez uniquement utiliser le paramètre `S3ObjectVersion` pour les objets soumis à la gestion des versions.

```
aws lambda create-function --function-name myFunction \
```

```
--runtime ruby3.2 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

Pour mettre à jour une fonction existante à l'aide de l'interface de ligne de commande, vous devez spécifier le nom de votre fonction à l'aide du paramètre `--function-name`. Vous devez également spécifier l'emplacement du fichier .zip que vous souhaitez utiliser pour mettre à jour votre code de fonction. Si votre fichier .zip se trouve dans un dossier sur votre machine de génération locale, utilisez l'option `--zip-file` pour spécifier le chemin d'accès du fichier, comme le montre l'exemple de commande suivant.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Pour spécifier l'emplacement du fichier .zip dans un compartiment Amazon S3, utilisez les options `--s3-bucket` et `--s3-key` comme le montre l'exemple de commande suivant. Vous devez uniquement utiliser le paramètre `--s3-object-version` pour les objets soumis à la gestion des versions.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Création et mise à jour de fonctions avec des fichiers .zip à l'aide de l'API Lambda

Pour créer et mettre à jour des fonctions à l'aide d'une archive de fichiers .zip, utilisez les opérations d'API suivantes :

- CreateFunction
- UpdateFunctionCode

Création et mise à jour de fonctions avec des fichiers .zip à l'aide d'AWS SAM

L'AWS Serverless Application Model (AWS SAM) est une boîte à outils qui permet de rationaliser le processus de création et d'exécution d'applications sans serveur sur AWS. Vous définissez les ressources de votre application dans un modèle YAML ou JSON et vous utilisez l'interface de ligne de commande AWS SAM (AWS SAM CLI) pour créer, empaqueter et déployer vos applications. Lorsque vous créez une fonction Lambda à partir d'un modèle AWS SAM, AWS SAM

crée automatiquement un package de déploiement .zip ou une image de conteneur avec le code de votre fonction et les dépendances que vous spécifiez. Pour en savoir plus sur l'utilisation d'AWS SAM pour créer et déployer des fonctions Lambda, veuillez consulter la rubrique Premiers pas avec AWS SAM dans le Guide du développeur AWS Serverless Application Model.

Vous pouvez également utiliser AWS SAM pour créer une fonction Lambda à l'aide d'une archive de fichiers .zip existante. Pour créer une fonction Lambda à l'aide d'AWS SAM, vous pouvez enregistrer votre fichier .zip dans un compartiment Amazon S3 ou dans un dossier local sur votre machine de génération. Pour savoir comment charger un fichier dans un compartiment Amazon S3 à l'aide de l'AWS CLI, veuillez consulter la rubrique Déplacer des objets dans le Guide de l'utilisateur AWS CLI.

Dans votre modèle AWS SAM, la ressource `AWS::Serverless::Function` spécifie votre fonction Lambda. Dans cette ressource, définissez les propriétés suivantes pour créer une fonction à l'aide d'une archive de fichiers .zip :

- `PackageType` : défini sur `Zip`
- `CodeUri` : défini comme l'URI Amazon S3 du code de la fonction, le chemin d'accès vers le dossier local ou l'objet FunctionCode
- `Runtime` : défini sur votre exécution choisie

Avec AWS SAM, si votre fichier .zip est supérieur à 50 Mo, vous n'avez pas à le charger au préalable dans un compartiment Amazon S3. AWS SAM peut charger des packages .zip jusqu'à la taille maximale autorisée de 250 Mo (décompressés) à partir d'un emplacement sur votre machine de génération locale.

Pour en savoir plus sur le déploiement de fonctions à l'aide d'un fichier .zip dans AWS SAM, veuillez consulter AWS::Serverless::Function dans le Guide du développeur AWS SAM.

Création et mise à jour de fonctions avec des fichiers .zip à l'aide d'CloudFormation

Vous pouvez utiliser CloudFormation pour créer une fonction Lambda à l'aide d'une archive de fichiers .zip. Pour créer une fonction Lambda à partir d'un fichier .zip, vous devez d'abord charger votre fichier dans un compartiment Amazon S3. Pour savoir comment charger un fichier dans un compartiment Amazon S3 à l'aide de l'AWS CLI, veuillez consulter la rubrique Déplacer des objets dans le Guide de l'utilisateur AWS CLI.

Dans votre modèle CloudFormation, la ressource `AWS::Lambda::Function` spécifie votre fonction Lambda. Dans cette ressource, définissez les propriétés suivantes pour créer une fonction à l'aide d'une archive de fichiers .zip :

- `PackageType` : défini sur `Zip`
- `Code` : saisissez le nom du compartiment Amazon S3 et le nom du fichier .zip dans les champs `S3Bucket` et `S3Key`
- `Runtime` : défini sur votre exécution choisie

Le fichier .zip que CloudFormation génère ne peut pas dépasser 4 Mo. Pour en savoir plus sur le déploiement de fonctions à l'aide d'un fichier .zip dans CloudFormation, veuillez consulter AWS::Lambda::Function dans le Guide de l'utilisateur CloudFormation.

Déploiement de fonctions Lambda en Ruby avec des images conteneurs

Il existe trois méthodes pour créer une image de conteneur pour une fonction Lambda Ruby :

- Utiliser une image AWS de base pour Ruby

  Les images de base AWS sont préchargées avec une exécution du langage, un client d'interface d'exécution pour gérer l'interaction entre Lambda et votre code de fonction, et un émulateur d'interface d'exécution pour les tests locaux.

- Utilisation d'une image de base AWS uniquement pour le système d'exploitation

  AWS Les images de base réservées au système d'exploitation contiennent une distribution Amazon Linux et l'émulateur d'interface d'exécution. Ces images sont couramment utilisées pour créer des images de conteneur pour les langages compilés, tels que Go et Rust, et pour une langue ou une version linguistique pour laquelle Lambda ne fournit pas d'image de base, comme Node.js 19. Vous pouvez également utiliser des images de base uniquement pour le système d'exploitation pour implémenter un environnement d'exécution personnalisé. Pour rendre l'image compatible avec Lambda, vous devez inclure le client d'interface d'exécution pour Ruby dans l'image.

- Utilisation d'une image non AWS basique

  Vous pouvez utiliser une autre image de base à partir d'un autre registre de conteneur, comme Alpine Linux ou Debian. Vous pouvez également utiliser une image personnalisée créée par votre organisation. Pour rendre l'image compatible avec Lambda, vous devez inclure le client d'interface d'exécution pour Ruby dans l'image.

> **Tip**
>
> Pour réduire le temps nécessaire à l'activation des fonctions du conteneur Lambda, consultez Utiliser des générations en plusieurs étapes (français non garanti) dans la documentation Docker. Pour créer des images de conteneur efficaces, suivez la section Bonnes pratiques pour l'écriture de Dockerfiles (français non garanti).

Cette page explique comment créer, tester et déployer des images de conteneur pour Lambda.

Rubriques

- AWS images de base pour Ruby
- Utiliser une image AWS de base pour Ruby
- Utilisation d'une autre image de base avec le client d'interface d'exécution

AWS images de base pour Ruby

AWS fournit les images de base suivantes pour Ruby :

Étiquettes	Environnement d'exécution	Système d'exploitation	Dockerfile	Obsolescence
3.4	Ruby 3.4	Amazon Linux	Dockerfile pour Ruby 3.4 sur GitHub	31 mars 2028
3.3	Ruby 3.3	Amazon Linux	Dockerfile pour Ruby 3.3 sur GitHub	31 mars 2027
3.2	Ruby 3.2	Amazon Linux 2	Dockerfile pour Ruby 3.2 sur GitHub	31 mars 2026

Référentiel Amazon ECR : gallery.ecr. aws/lambda/ruby

Utiliser une image AWS de base pour Ruby

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- AWS CLI version 2
- Docker (version minimale 25.0.0)
- Le plug-in buildx Docker.
- Ruby

Création d'une image à partir d'une image de base

Pour créer une image de conteneur pour Ruby

1. Créez un répertoire pour le projet, puis passez à ce répertoire.

```
mkdir example
cd example
```

2. Créez un nouveau fichier appelé `Gemfile`. C'est ici que vous listez les RubyGems packages requis pour votre application. Le AWS SDK pour Ruby est disponible auprès de RubyGems. Vous devez choisir des gemmes AWS de service spécifiques à installer. Par exemple, pour utiliser la gem Ruby pour Lambda, votre Gemfile doit ressembler à ceci :

```
source 'https://rubygems.org'

gem 'aws-sdk-lambda'
```

   Sinon, la gemme aws-sdk contient toutes les gemmes de service disponibles. AWS Cette gem est très large. Nous vous recommandons de ne l'utiliser que si vous dépendez de nombreux AWS services.

3. Installez les dépendances spécifiées dans le fichier Gemfile à l'aide de l'installation groupée.

```
bundle install
```

4. Créez un nouveau fichier appelé `lambda_function.rb`. Vous pouvez ajouter l'exemple de code de fonction suivant au fichier pour le tester, ou utiliser le vôtre.

   Example Fonction Ruby

```
module LambdaFunction
  class Handler
    def self.process(event:,context:)
      "Hello from Lambda!"
    end
  end
end
```

5. Créez un nouveau fichier Docker. Voici un exemple de Dockerfile qui utilise une image de base AWS. Le Dockerfile utilise la configuration suivante :

- Définissez la propriété FROM sur l'URI de l'image de base.
- Utilisez la commande COPY pour copier le code de la fonction et les dépendances de l'environnement d'exécution dans {LAMBDA_TASK_ROOT}, une variable d'environnement définie par Lambda.
- Définir l'argument CMD pour le gestionnaire de la fonction Lambda.

Notez que l'exemple de Dockerfile n'inclut pas d'instruction USER. Lorsque vous déployez une image de conteneur sur Lambda, Lambda définit automatiquement un utilisateur Linux par défaut disposant d'autorisations de moindre privilège. Ceci est différent du comportement standard de Docker qui est défini par défaut par l'utilisateur `root` lorsqu'aucune instruction USER n'est fournie.

Example Dockerfile

```
FROM public.ecr.aws/lambda/ruby:3.4

# Copy Gemfile and Gemfile.lock
COPY Gemfile Gemfile.lock ${LAMBDA_TASK_ROOT}/

# Install Bundler and the specified gems
RUN gem install bundler:2.4.20 && \
    bundle config set --local path 'vendor/bundle' && \
    bundle install

# Copy function code
COPY lambda_function.rb ${LAMBDA_TASK_ROOT}/

# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "lambda_function.LambdaFunction::Handler.process" ]
```

6. Générez l'image Docker à l'aide de la commande docker build. L'exemple suivant nomme l'image `docker-image` et lui donne la balise `test`. Pour rendre votre image compatible avec Lambda, vous devez utiliser l'option `--provenance=false`.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> La commande spécifie l'option `--platform linux/amd64` pour garantir la compatibilité de votre conteneur avec l'environnement d'exécution Lambda, quelle que soit l'architecture de votre machine de génération. Si vous avez l'intention de créer une fonction Lambda à l'aide de l'architecture du jeu ARM64 d'instructions, veillez à modifier la commande pour utiliser l'`--platform linux/arm64`option à la place.

(Facultatif) Testez l'image localement

1. Démarrez votre image Docker à l'aide de la commande docker run. Dans cet exemple, `docker-image` est le nom de l'image et `test` est la balise.

   ```
   docker run --platform linux/amd64 -p 9000:8080 docker-image:test
   ```

   Cette commande exécute l'image en tant que conteneur et crée un point de terminaison local à `localhost:9000/2015-03-31/functions/function/invocations`.

   > **Note**
   >
   > Si vous avez créé l'image Docker pour l'architecture du jeu ARM64 d'instructions, veillez à utiliser l'`--platform linux/arm64`option au lieu de`--platform linux/amd64`.

2. À partir d'une nouvelle fenêtre de terminal, publiez un événement au point de terminaison local.

   Linux/macOS

   Sous Linux et macOS, exécutez la commande `curl` suivante :

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   Cette commande invoque la fonction avec un événement vide et renvoie une réponse. Si vous utilisez votre propre code de fonction plutôt que l'exemple de code de fonction, vous pouvez invoquer la fonction avec une charge utile JSON. Exemple :

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

Dans PowerShell, exécutez la `Invoke-WebRequest` commande suivante :

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Cette commande invoque la fonction avec un événement vide et renvoie une réponse. Si vous utilisez votre propre code de fonction plutôt que l'exemple de code de fonction, vous pouvez invoquer la fonction avec une charge utile JSON. Exemple :

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. Obtenez l'ID du conteneur.

```
docker ps
```

4. Utilisez la commande docker kill pour arrêter le conteneur. Dans cette commande, remplacez `3766c4ab331c` par l'ID du conteneur de l'étape précédente.

```
docker kill 3766c4ab331c
```

Déploiement de l'image

Pour charger l'image sur Amazon RIE et créer la fonction Lambda

1. Exécutez la get-login-passwordcommande pour authentifier la CLI Docker auprès de votre registre Amazon ECR.
   - Définissez la `--region` valeur à l' Région AWS endroit où vous souhaitez créer le référentiel Amazon ECR.
   - `111122223333`Remplacez-le par votre Compte AWS identifiant.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Créez un référentiel dans Amazon ECR à l'aide de la commande create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Le référentiel Amazon ECR doit être Région AWS identique à la fonction Lambda.

En cas de succès, vous obtenez une réponse comme celle-ci :

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Copiez le `repositoryUri` à partir de la sortie de l'étape précédente.
4. Exécutez la commande docker tag pour étiqueter votre image locale dans votre référentiel Amazon ECR en tant que dernière version. Dans cette commande :

- `docker-image:test` est le nom et la balise de votre image Docker. Il s'agit du nom et de la balise de l'image que vous avez spécifiés dans la commande `docker build`.
- Remplacez `<ECRrepositoryUri>` par l'`repositoryUri` que vous avez copié. Assurez-vous d'inclure `:latest` à la fin de l'URI.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Exemple :

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Exécutez la commande docker push pour déployer votre image locale dans le référentiel Amazon ECR. Assurez-vous d'inclure `:latest` à la fin de l'URI du référentiel.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Créez un rôle d'exécution pour la fonction, si vous n'en avez pas déjà un. Vous aurez besoin de l'Amazon Resource Name (ARN) du rôle à l'étape suivante.
7. Créez la fonction Lambda. Pour `ImageUri`, indiquez l'URI du référentiel mentionné précédemment. Assurez-vous d'inclure `:latest` à la fin de l'URI.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> Vous pouvez créer une fonction en utilisant une image d'un autre AWS compte, à condition que l'image se trouve dans la même région que la fonction Lambda. Pour de plus amples informations, veuillez consulter Autorisations entre comptes Amazon ECR.

8. Invoquer la fonction.

```
aws lambda invoke --function-name hello-world response.json
```

Vous devriez obtenir une réponse comme celle-ci :

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Pour voir la sortie de la fonction, consultez le fichier `response.json`.

Pour mettre à jour le code de fonction, vous devez créer à nouveau l'image, télécharger la nouvelle image dans le référentiel Amazon ECR, puis utiliser la update-function-codecommande pour déployer l'image sur la fonction Lambda.

Lambda résout l'étiquette d'image en hachage d'image spécifique. Cela signifie que si vous pointez la balise d'image qui a été utilisée pour déployer la fonction vers une nouvelle image dans Amazon ECR, Lambda ne met pas automatiquement à jour la fonction pour utiliser la nouvelle image.

Pour déployer la nouvelle image sur la même fonction Lambda, vous devez utiliser la update-function-codecommande, même si la balise d'image dans Amazon ECR reste la même. Dans l'exemple suivant, l'option `--publish` crée une version de la fonction à l'aide de l'image du conteneur mise à jour.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Utilisation d'une autre image de base avec le client d'interface d'exécution

Si vous utilisez une image de base uniquement pour le système d'exploitation ou une autre image de base, vous devez inclure le client d'interface d'exécution dans votre image. Le client d'interface d'exécution étend leAPI de runtime, qui gère l'interaction entre Lambda et votre code de fonction.

Installez le client d'interface d'exécution Lambda pour Ruby à l'aide du gestionnaire de packages RubyGems .org :

```
gem install aws_lambda_ric
```

Vous pouvez également télécharger le client d'interface d'exécution Ruby depuis GitHub.

L'exemple suivant montre comment créer une image de conteneur pour Ruby à l'aide d'une image non AWS basique. L'exemple Dockerfile utilise une image de base Ruby officielle. Le Dockerfile inclut le client d'interface d'exécution.

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- AWS CLI version 2
- Docker (version minimale 25.0.0)
- Le plug-in buildx Docker.
- Ruby

Création d'une image à partir d'une image de base alternative

Pour créer une image de conteneur pour Ruby au moyen d'une autre image de base

1. Créez un répertoire pour le projet, puis passez à ce répertoire.

   ```
   mkdir example
   cd example
   ```

2. Créez un nouveau fichier appelé `Gemfile`. C'est ici que vous listez les RubyGems packages requis pour votre application. Le AWS SDK pour Ruby est disponible auprès de RubyGems. Vous devez choisir des gemmes AWS de service spécifiques à installer. Par exemple, pour utiliser la gem Ruby pour Lambda, votre Gemfile doit ressembler à ceci :

   ```
   source 'https://rubygems.org'

   gem 'aws-sdk-lambda'
   ```

   Sinon, la gemme aws-sdk contient toutes les gemmes de service disponibles. AWS Cette gem est très large. Nous vous recommandons de ne l'utiliser que si vous dépendez de nombreux AWS services.

3. Installez les dépendances spécifiées dans le fichier Gemfile à l'aide de l'installation groupée.

```
bundle install
```

4. Créez un nouveau fichier appelé `lambda_function.rb`. Vous pouvez ajouter l'exemple de code de fonction suivant au fichier pour le tester, ou utiliser le vôtre.

   Example Fonction Ruby

```
module LambdaFunction
  class Handler
    def self.process(event:,context:)
      "Hello from Lambda!"
    end
  end
end
```

5. Créez un nouveau fichier Docker. Le Dockerfile suivant utilise une image de base Ruby au lieu d'une image de base AWS. Le Dockerfile inclut le client d'interface d'exécution pour Ruby, ce qui rend l'image compatible avec Lambda. Vous pouvez également ajouter le client de l'interface d'exécution au Gemfile de votre application.

   - Définissez la propriété `FROM` sur l'image de base Ruby.
   - Créez un répertoire pour le code de fonction et une variable d'environnement pointant vers ce répertoire. Dans cet exemple, le répertoire est `/var/task`, qui reflète l'environnement d'exécution Lambda. Cependant, vous pouvez choisir n'importe quel répertoire pour le code de fonction car le Dockerfile n'utilise pas d'image de AWS base.
   - Définissez le `ENTRYPOINT` sur le module que vous souhaitez que le conteneur Docker exécute lorsqu'il démarre. Dans ce cas, le module est le client d'interface d'exécution.
   - Définir l'argument `CMD` pour le gestionnaire de la fonction Lambda.

   Notez que l'exemple de Dockerfile n'inclut pas d'instruction USER. Lorsque vous déployez une image de conteneur sur Lambda, Lambda définit automatiquement un utilisateur Linux par défaut disposant d'autorisations de moindre privilège. Ceci est différent du comportement standard de Docker qui est défini par défaut par l'utilisateur `root` lorsqu'aucune instruction `USER` n'est fournie.

Example Dockerfile

```
FROM ruby:2.7

# Install the runtime interface client for Ruby
RUN gem install aws_lambda_ric

# Add the runtime interface client to the PATH
ENV PATH="/usr/local/bundle/bin:${PATH}"

# Create a directory for the Lambda function
ENV LAMBDA_TASK_ROOT=/var/task
RUN mkdir -p ${LAMBDA_TASK_ROOT}
WORKDIR ${LAMBDA_TASK_ROOT}

# Copy Gemfile and Gemfile.lock
COPY Gemfile Gemfile.lock ${LAMBDA_TASK_ROOT}/

# Install Bundler and the specified gems
RUN gem install bundler:2.4.20 && \
    bundle config set --local path 'vendor/bundle' && \
    bundle install

# Copy function code
COPY lambda_function.rb ${LAMBDA_TASK_ROOT}/

# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "aws_lambda_ric" ]

# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "lambda_function.LambdaFunction::Handler.process" ]
```

6. Générez l'image Docker à l'aide de la commande docker build. L'exemple suivant nomme l'image `docker-image` et lui donne la balise `test`. Pour rendre votre image compatible avec Lambda, vous devez utiliser l'option `--provenance=false`.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> La commande spécifie l'option `--platform linux/amd64` pour garantir la compatibilité de votre conteneur avec l'environnement d'exécution Lambda, quelle que soit l'architecture de votre machine de génération. Si vous avez l'intention de créer une fonction Lambda à l'aide de l'architecture du jeu ARM64 d'instructions, veillez à modifier la commande pour utiliser l'`--platform linux/arm64`option à la place.

(Facultatif) Testez l'image localement

Utilisez l'émulateur d'interface d'exécution pour tester l'image localement. Vous pouvez intégrer l'émulateur dans votre image ou utiliser la procédure suivante pour l'installer sur votre machine locale.

Pour installer et exécuter l'émulateur d'interface d'exécution sur votre ordinateur local

1. Depuis le répertoire de votre projet, exécutez la commande suivante pour télécharger l'émulateur d'interface d'exécution (architecture x86-64) GitHub et l'installer sur votre machine locale.

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   Pour installer l'émulateur arm64, remplacez l'URL du GitHub référentiel dans la commande précédente par la suivante :

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

   PowerShell

   ```
   $dirPath = "$HOME\.aws-lambda-rie"
   if (-not (Test-Path $dirPath)) {
       New-Item -Path $dirPath -ItemType Directory
   }
   ```

```
$downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/
releases/latest/download/aws-lambda-rie"
$destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

Pour installer l'émulateur arm64, remplacez `$downloadLink` par ce qui suit :

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. Démarrez votre image Docker à l'aide de la commande docker run. Notez ce qui suit :

   - `docker-image` est le nom de l'image et `test` est la balise.
   - `aws_lambda_ric lambda_function.LambdaFunction::Handler.process` est le `ENTRYPOINT` suivi du `CMD` depuis votre Dockerfile.

   Linux/macOS

   ```
   docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
    9000:8080 \
       --entrypoint /aws-lambda/aws-lambda-rie \
       docker-image:test \
           aws_lambda_ric lambda_function.LambdaFunction::Handler.process
   ```

   PowerShell

   ```
   docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
    9000:8080 `
   --entrypoint /aws-lambda/aws-lambda-rie `
   docker-image:test `
       aws_lambda_ric lambda_function.LambdaFunction::Handler.process
   ```

   Cette commande exécute l'image en tant que conteneur et crée un point de terminaison local à `localhost:9000/2015-03-31/functions/function/invocations`.

> **Note**
>
> Si vous avez créé l'image Docker pour l'architecture du jeu ARM64 d'instructions, veillez à utiliser l'`--platform linux/arm64`option au lieu de`--platform linux/amd64`.

3. Publiez un événement au point de terminaison local.

   Linux/macOS

   Sous Linux et macOS, exécutez la commande `curl` suivante :

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   Cette commande invoque la fonction avec un événement vide et renvoie une réponse. Si vous utilisez votre propre code de fonction plutôt que l'exemple de code de fonction, vous pouvez invoquer la fonction avec une charge utile JSON. Exemple :

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   Dans PowerShell, exécutez la `Invoke-WebRequest` commande suivante :

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```

   Cette commande invoque la fonction avec un événement vide et renvoie une réponse. Si vous utilisez votre propre code de fonction plutôt que l'exemple de code de fonction, vous pouvez invoquer la fonction avec une charge utile JSON. Exemple :

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
    "application/json"
   ```

4. Obtenez l'ID du conteneur.

   ```
   docker ps
   ```

5. Utilisez la commande docker kill pour arrêter le conteneur. Dans cette commande, remplacez `3766c4ab331c` par l'ID du conteneur de l'étape précédente.

```
docker kill 3766c4ab331c
```

Déploiement de l'image

Pour charger l'image sur Amazon RIE et créer la fonction Lambda

1. Exécutez la get-login-passwordcommande pour authentifier la CLI Docker auprès de votre registre Amazon ECR.
   - Définissez la `--region` valeur à l' Région AWS endroit où vous souhaitez créer le référentiel Amazon ECR.
   - `111122223333`Remplacez-le par votre Compte AWS identifiant.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Créez un référentiel dans Amazon ECR à l'aide de la commande create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Le référentiel Amazon ECR doit être Région AWS identique à la fonction Lambda.

En cas de succès, vous obtenez une réponse comme celle-ci :

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
```

```
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Copiez le `repositoryUri` à partir de la sortie de l'étape précédente.
4. Exécutez la commande docker tag pour étiqueter votre image locale dans votre référentiel Amazon ECR en tant que dernière version. Dans cette commande :

   - `docker-image:test` est le nom et la balise de votre image Docker. Il s'agit du nom et de la balise de l'image que vous avez spécifiés dans la commande `docker build`.
   - Remplacez <ECRrepositoryUri> par l'`repositoryUri` que vous avez copié. Assurez-vous d'inclure `:latest` à la fin de l'URI.

   ```
   docker tag docker-image:test <ECRrepositoryUri>:latest
   ```

   Exemple :

   ```
   docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
   world:latest
   ```

5. Exécutez la commande docker push pour déployer votre image locale dans le référentiel Amazon ECR. Assurez-vous d'inclure `:latest` à la fin de l'URI du référentiel.

   ```
   docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

6. Créez un rôle d'exécution pour la fonction, si vous n'en avez pas déjà un. Vous aurez besoin de l'Amazon Resource Name (ARN) du rôle à l'étape suivante.
7. Créez la fonction Lambda. Pour `ImageUri`, indiquez l'URI du référentiel mentionné précédemment. Assurez-vous d'inclure `:latest` à la fin de l'URI.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> Vous pouvez créer une fonction en utilisant une image d'un autre AWS compte, à condition que l'image se trouve dans la même région que la fonction Lambda. Pour de plus amples informations, veuillez consulter Autorisations entre comptes Amazon ECR.

8. Invoquer la fonction.

```
aws lambda invoke --function-name hello-world response.json
```

Vous devriez obtenir une réponse comme celle-ci :

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Pour voir la sortie de la fonction, consultez le fichier `response.json`.

Pour mettre à jour le code de fonction, vous devez créer à nouveau l'image, télécharger la nouvelle image dans le référentiel Amazon ECR, puis utiliser la update-function-codecommande pour déployer l'image sur la fonction Lambda.

Lambda résout l'étiquette d'image en hachage d'image spécifique. Cela signifie que si vous pointez la balise d'image qui a été utilisée pour déployer la fonction vers une nouvelle image dans Amazon ECR, Lambda ne met pas automatiquement à jour la fonction pour utiliser la nouvelle image.

Pour déployer la nouvelle image sur la même fonction Lambda, vous devez utiliser la update-function-codecommande, même si la balise d'image dans Amazon ECR reste la même. Dans l'exemple suivant, l'option `--publish` crée une version de la fonction à l'aide de l'image du conteneur mise à jour.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Utilisation de couches pour les fonctions Lambda Ruby

Utilisez les couches Lambda pour empaqueter le code et les dépendances que vous souhaitez réutiliser dans plusieurs fonctions. Les couches contiennent généralement des dépendances de bibliothèque, une exécution personnalisée, ou des fichiers de configuration. La création d'une couche implique trois étapes générales :

1. Empaquetez le contenu de votre couche. Cela signifie créer une archive de fichiers .zip contenant les dépendances que vous souhaitez utiliser dans vos fonctions.
2. Créez la couche dans Lambda.
3. Ajoutez la couche à vos fonctions.

Rubriques

- Empaqueter le contenu de votre couche
- Créer la couche dans Lambda
- Utiliser des gems provenant de couches dans une fonction
- Ajouter la couche à votre fonction
- Exemple d'application

Empaqueter le contenu de votre couche

Pour créer une couche, regroupez vos packages dans une archive (fichier .zip) répondant aux exigences suivantes :

- Créez la couche à l'aide de la même version de Ruby que celle que vous prévoyez d'utiliser pour la fonction Lambda. Par exemple, si vous créez votre couche pour Ruby 3.4, utilisez l'environnement d'exécution Ruby 3.4 pour votre fonction.
- Le fichier .zip de votre couche doit utiliser l'une des structures de répertoires suivantes :
  - `ruby/gems/`*`x.x.x`* (où *`x.x.x`* est votre version de Ruby, par exemple `3.4.0`)
  - `ruby/lib`

  Pour de plus amples informations, consultez Chemins d'accès de couche pour chaque exécution Lambda.
- Les packages de votre couche doivent être compatibles avec Linux. Les fonctions Lambda s'exécutent sur Amazon Linux.

Vous pouvez créer des couches contenant des gems Ruby tierces ou vos propres modules et classes Ruby. De nombreuses gems Ruby populaires contiennent des extensions natives (code C) qui doivent être compilées pour l'environnement Lambda Linux.

Gems Ruby pures

Les gems Ruby pures contiennent uniquement du code Ruby et ne nécessitent pas de compilation. Ces gems sont plus simples à empaqueter et à utiliser sur différentes plateformes.

Créer une couche à l'aide de gems Ruby pures

1. Créez un `Gemfile` pour spécifier les gems Ruby pures que vous souhaitez inclure dans votre couche :

   Example Gemfile

   ```
   source 'https://rubygems.org'

   gem 'tzinfo'
   ```

2. Installez les gems dans le répertoire `vendor/bundle` à l'aide de Bundler :

   ```
   bundle config set --local path vendor/bundle
   bundle install
   ```

3. Copiez les gems installées dans la structure de répertoires requise par Lambda (`ruby/gems/3.4.0`) :

   ```
   mkdir -p ruby/gems/3.4.0
   cp -r vendor/bundle/ruby/3.4.0*/* ruby/gems/3.4.0/
   ```

4. Compressez le contenu de la couche :

   Linux/macOS

   ```
   zip -r layer.zip ruby/
   ```

   PowerShell

   ```
   Compress-Archive -Path .\ruby -DestinationPath .\layer.zip
   ```

La structure de répertoire de votre fichier .zip devrait se présenter comme suit :

```
ruby/
### gems/
    ### 3.4.0/
        ### gems/
        #   ### concurrent-ruby-1.3.5/
        #   ### tzinfo-2.0.6/
        ### specifications/
        ### cache/
        ### build_info/
        ### (other bundler directories)
```

> **Note**
>
> Vous devez requérir chaque gem individuellement dans le code de votre fonction. Vous ne pouvez pas utiliser `bundler/setup` ou `Bundler.require`. Pour de plus amples informations, consultez Utiliser des gems provenant de couches dans une fonction.

Gems avec des extensions natives

De nombreuses gems Ruby populaires contiennent des extensions natives (code C) qui doivent être compilées pour la plateforme cible. Les gems populaires dotées d'extensions natives incluent nokogiri, pg, mysql2, sqlite3, and ffi. Ces gems doivent être créées dans un environnement Linux compatible avec l'environnement d'exécution Lambda.

Créer une couche à l'aide de gems avec des extensions natives

1. Créer un de `Gemfile`.

   Example Gemfile

   ```
   source 'https://rubygems.org'

   gem 'nokogiri'
   gem 'httparty'
   ```

2. Utilisez Docker pour créer les gems dans un environnement Linux compatible avec Lambda. Spécifiez une image de base AWS dans votre Dockerfile :

   Example Dockerfile pour Ruby 3.4

```
FROM public.ecr.aws/lambda/ruby:3.4

# Copy Gemfile
COPY Gemfile ./

# Install system dependencies for native extensions
RUN dnf update -y && \
    dnf install -y gcc gcc-c++ make

# Configure bundler and install gems
RUN bundle config set --local path vendor/bundle && \
    bundle install

# Create the layer structure
RUN mkdir -p ruby/gems/3.4.0 && \
    cp -r vendor/bundle/ruby/3.4.0*/* ruby/gems/3.4.0/

# Create the layer zip file
RUN zip -r layer.zip ruby/
```

3. Créez l'image et extrayez la couche :

```
docker build -t ruby-layer-builder .
docker run --rm -v $(pwd):/output --entrypoint cp ruby-layer-builder layer.zip /
output/
```

   Cela crée les gems dans l'environnement Linux approprié et copie le fichier `layer.zip` dans votre répertoire local. La structure de répertoire de votre fichier .zip devrait se présenter comme suit :

```
ruby/
### gems/
    ### 3.4.0/
        ### gems/
        #   ### bigdecimal-3.2.2/
        #   ### csv-3.3.5/
        #   ### httparty-0.23.1/
```

```
#   ### mini_mime-1.1.5/
#   ### multi_xml-0.7.2/
#   ### nokogiri-1.18.8-x86_64-linux-gnu/
#   ### racc-1.8.1/
### build_info/
### cache/
### specifications/
### (other bundler directories)
```

> **Note**
>
> Vous devez requérir chaque gem individuellement dans le code de votre fonction. Vous ne pouvez pas utiliser `bundler/setup` ou `Bundler.require`. Pour de plus amples informations, consultez Utiliser des gems provenant de couches dans une fonction.

Modules Ruby personnalisés

Créer une couche à l'aide de votre propre code

1. Créez la structure de répertoires requise pour votre couche :

   ```
   mkdir -p ruby/lib
   ```

2. Créez vos modules Ruby dans le répertoire `ruby/lib`. L'exemple de module suivant valide les commandes en confirmant qu'elles contiennent les informations requises.

   Example ruby/lib/order_validator.rb

   ```
   require 'json'

   module OrderValidator
     class ValidationError < StandardError; end

     def self.validate_order(order_data)
       # Validates an order and returns formatted data
       required_fields = %w[product_id quantity]

       # Check required fields
       missing_fields = required_fields.reject { |field| order_data.key?(field) }
       unless missing_fields.empty?
   ```

```
      raise ValidationError, "Missing required fields: #{missing_fields.join(',
 ')}"
    end

    # Validate quantity
    quantity = order_data['quantity']
    unless quantity.is_a?(Integer) && quantity > 0
      raise ValidationError, 'Quantity must be a positive integer'
    end

    # Format and return the validated data
    {
      'product_id' => order_data['product_id'].to_s,
      'quantity' => quantity,
      'shipping_priority' => order_data.fetch('priority', 'standard')
    }
  end

  def self.format_response(status_code, body)
    # Formats the API response
    {
      statusCode: status_code,
      body: JSON.generate(body)
    }
  end
end
```

3. Compressez le contenu de la couche :

   Linux/macOS

   ```
   zip -r layer.zip ruby/
   ```

   PowerShell

   ```
   Compress-Archive -Path .\ruby -DestinationPath .\layer.zip
   ```

   La structure de répertoire de votre fichier .zip devrait se présenter comme suit :

   ```
   ruby/
   ### lib/
   ```

```
    ### order_validator.rb
```

4. Dans votre fonction, exigez (require) et utilisez les modules. Vous devez requérir chaque gem individuellement dans le code de votre fonction. Vous ne pouvez pas utiliser `bundler/setup` ou `Bundler.require`. Pour de plus amples informations, consultez Utiliser des gems provenant de couches dans une fonction. Exemple :

```
require 'json'
require 'order_validator'

def lambda_handler(event:, context:)
  begin
    # Parse the order data from the event body
    order_data = JSON.parse(event['body'] || '{}')

    # Validate and format the order
    validated_order = OrderValidator.validate_order(order_data)

    OrderValidator.format_response(200, {
      message: 'Order validated successfully',
      order: validated_order
    })
  rescue OrderValidator::ValidationError => e
    OrderValidator.format_response(400, {
      error: e.message
    })
  rescue => e
    OrderValidator.format_response(500, {
      error: 'Internal server error'
    })
  end
end
```

Vous pouvez utiliser l'événement de test suivant afin d'invoquer la fonction :

```
{
    "body": "{\"product_id\": \"ABC123\", \"quantity\": 2, \"priority\": \"express
\"}"
}
```

Réponse attendue :

```
{
  "statusCode": 200,
  "body": "{\"message\":\"Order validated successfully\",\"order\":{\"product_id\":
\"ABC123\",\"quantity\":2,\"shipping_priority\":\"express\"}}"
}
```

Créer la couche dans Lambda

Vous pouvez publier votre couche en utilisant AWS CLI ou la console Lambda.

AWS CLI

Exécutez la commande AWS CLI publish-layer-version pour créer la couche Lambda :

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip
 --compatible-runtimes ruby3.4
```

Le paramètre compatible runtimes est facultatif. Lorsqu'il est spécifié, Lambda utilise ce paramètre pour filtrer les couches dans la console Lambda.

Console

Pour créer une couche (console)

1. Ouvrez la page Couches de la console Lambda.
2. Sélectionnez Créer un calque.
3. Choisissez Charger un fichier .zip, puis chargez l'archive .zip que vous avez créée précédemment.
4. (Facultatif) Pour Environnements d'exécution compatibles, choisissez l'environnement d'exécution Ruby qui correspond à la version de Ruby que vous avez utilisée pour créer votre couche.
5. Choisissez Créer.

Utiliser des gems provenant de couches dans une fonction

Dans le code de votre fonction, vous devez explicitement exiger chaque gem que vous souhaitez utiliser. Les commandes Bundler comme `bundler/setup` et `Bundler.require` ne sont pas prises en charge. Voici comment utiliser correctement les gems d'une couche dans une fonction Lambda :

```
# Correct: Use explicit requires for each gem
require 'nokogiri'
require 'httparty'

def lambda_handler(event:, context:)
  # Use the gems directly
  doc = Nokogiri::HTML(event['html'])
  response = HTTParty.get(event['url'])
  # ... rest of your function
end

# Incorrect: These Bundler commands will not work
# require 'bundler/setup'
# Bundler.require
```

Ajouter la couche à votre fonction

AWS CLI

Pour associer la couche à votre fonction, exécutez la commande AWS CLI update-function-configuration. Utilisez l'ARN de la couche pour le paramètre `--layers`. L'ARN doit spécifier la version (par exemple, `arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`). Pour de plus amples informations, consultez Couches et versions de couches.

```
aws lambda update-function-configuration --function-name my-function --cli-binary-format raw-in-base64-out --layers "arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1"
```

L'option cli-binary-format est obligatoire si vous utilisez AWS CLI version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

Console

Ajouter une couche à une fonction

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez la fonction.
3. Faites défiler jusqu'à la section Couches, puis choisissez Ajouter une couche.
4. Sous Choisir une couche, sélectionnez Couches personnalisées, puis choisissez votre couche.

> **Note**
>
> Si vous n'avez pas ajouté d'environnement d'exécution compatible lors de la création de la couche, celle-ci ne sera pas répertoriée ici. Vous pouvez spécifier l'ARN de la couche à la place.

5. Choisissez Ajouter.

Exemple d'application

Pour d'autres exemples d'utilisation des couches Lambda, consultez l'exemple d'application layer-ruby dans le référentiel GitHub du Guide du développeur AWS Lambda. Cette application inclut une couche qui contient la bibliothèque tzinfo. Après avoir créé la couche, vous pouvez déployer et invoquer la fonction correspondante pour confirmer que la couche fonctionne comme prévu.

Utilisation de l'objet de contexte Lambda pour récupérer les informations de la fonction Ruby

Lorsque Lambda exécute votre fonction, il transmet un objet contexte au gestionnaire. Cet objet fournit des méthodes et des propriétés fournissant des informations sur l'invocation, la fonction et l'environnement d'exécution.

Méthodes de contexte

- `get_remaining_time_in_millis` – Renvoie le nombre de millisecondes restant avant l'expiration de l'exécution.

Propriétés du contexte

- `function_name` – Nom de la fonction Lambda.
- `function_version` – Version de la fonction.
- `invoked_function_arn` – Amazon Resource Name (ARN) utilisé pour appeler la fonction. Indique si l'appelant a spécifié un numéro de version ou un alias.
- `memory_limit_in_mb` – Quantité de mémoire allouée à la fonction.
- `aws_request_id` – Identifiant de la demande d'invocation.
- `log_group_name` – Groupe de journaux pour la fonction.
- `log_stream_name` – Flux de journal de l'instance de fonction.
- `deadline_ms` – Date d'expiration de l'exécution, exprimée en millisecondes au format horaire Unix.
- `identity` – (applications mobiles) Informations sur l'identité Amazon Cognito qui a autorisé la demande.
- `client_context` – (applications mobiles) Contexte client fourni à Lambda par l'application client.

Journalisation et surveillance des fonctions Lambda Ruby

AWS Lambda surveille automatiquement les fonctions Lambda en votre nom et envoie les journaux à Amazon. CloudWatch Votre fonction Lambda est fournie avec un groupe de CloudWatch journaux Logs et un flux de journaux pour chaque instance de votre fonction. L'environnement d'exécution Lambda envoie des informations sur chaque invocation au flux de journaux et transmet les journaux et autres sorties provenant du code de votre fonction. Pour de plus amples informations, veuillez consulter Envoi automatique des journaux de la fonction Lambda à CloudWatch Logs.

Cette page explique comment générer une sortie de journal à partir du code de votre fonction Lambda et comment accéder aux journaux à l'aide de la AWS Command Line Interface console Lambda ou de la console. CloudWatch

Sections

- Création d'une fonction qui renvoie des journaux
- Affichage des journaux dans la console Lambda
- Afficher les journaux dans la CloudWatch console
- Afficher les journaux à l'aide de AWS Command Line Interface (AWS CLI)
- Suppression de journaux
- Utilisation de la bibliothèque Ruby logger

Création d'une fonction qui renvoie des journaux

Pour générer les journaux à partir de votre code de fonction, vous pouvez utiliser des déclarations `puts` ou n'importe quelle bibliothèque de journalisation qui écrit dans `stdout` ou `stderr`. L'exemple suivant consigne les valeurs des variables d'environnement et l'objet d'événement.

Example lambda_function.rb

```
# lambda_function.rb

def handler(event:, context:)
    puts "## ENVIRONMENT VARIABLES"
    puts ENV.to_a
    puts "## EVENT"
    puts event.to_a
```

```
end
```

Example format des journaux

```
START RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95 Version: $LATEST
## ENVIRONMENT VARIABLES
environ({'AWS_LAMBDA_LOG_GROUP_NAME': '/aws/lambda/my-function',
 'AWS_LAMBDA_LOG_STREAM_NAME': '2020/01/31/[$LATEST]3893xmpl7fac4485b47bb75b671a283c',
 'AWS_LAMBDA_FUNCTION_NAME': 'my-function', ...})
## EVENT
{'key': 'value'}
END RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95
REPORT RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95  Duration: 15.74 ms  Billed
 Duration: 147 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 130.49 ms
XRAY TraceId: 1-5e34a614-10bdxmplf1fb44f07bc535a1   SegmentId: 07f5xmpl2d1f6f85
 Sampled: true
```

L'exécution Ruby enregistre les lignes `START`, `END` et `REPORT` pour chaque invocation. La ligne de rapport fournit les détails suivants.

Champs de données de la ligne REPORT

- RequestId— L'identifiant de demande unique pour l'invocation.
- Duration – Temps que la méthode de gestion du gestionnaire de votre fonction a consacré au traitement de l'événement.
- Billed Duration : temps facturé pour l'invocation.
- Memory Size – Quantité de mémoire allouée à la fonction.
- Max Memory Used – Quantité de mémoire utilisée par la fonction. Lorsque les appels partagent un environnement d'exécution, Lambda indique la mémoire maximale utilisée pour toutes les invocations. Ce comportement peut entraîner une valeur signalée plus élevée que prévu.
- Init Duration : pour la première requête servie, temps qu'il a pris à l'exécution charger la fonction et exécuter le code en dehors de la méthode du gestionnaire.
- XRAY TraceId — Pour les demandes suivies, l'ID de AWS X-Ray trace.
- SegmentId— Pour les demandes tracées, l'identifiant du segment X-Ray.
- Sampled : pour les demandes suivies, résultat de l'échantillonnage.

Pour obtenir des journaux plus détaillés, utilisez la bibliothèque the section called "Utilisation de la bibliothèque Ruby logger".

Affichage des journaux dans la console Lambda

Vous pouvez utiliser la console Lambda pour afficher la sortie du journal après avoir invoqué une fonction Lambda.

Si votre code peut être testé à partir de l'éditeur Code intégré, vous trouverez les journaux dans les résultats d'exécution. Lorsque vous utilisez la fonctionnalité de test de console pour invoquer une fonction, vous trouverez Sortie du journal dans la section Détails.

Afficher les journaux dans la CloudWatch console

Vous pouvez utiliser la CloudWatch console Amazon pour consulter les journaux de toutes les invocations de fonctions Lambda.

Pour afficher les journaux sur la CloudWatch console

1. Ouvrez la page Groupes de journaux sur la CloudWatch console.
2. Choisissez le groupe de journaux pour votre fonction (***your-function-name***/aws/lambda/).
3. Choisissez un flux de journaux.

Chaque flux de journal correspond à une instance de votre fonction. Un flux de journal apparaît lorsque vous mettez à jour votre fonction Lambda et lorsque des instances supplémentaires sont créées pour gérer les appels simultanés. Pour trouver les journaux d'un appel spécifique, nous vous recommandons d'instrumenter votre fonction avec. AWS X-Ray X-Ray enregistre des détails sur la demande et le flux de journaux dans le suivi.

Afficher les journaux à l'aide de AWS Command Line Interface (AWS CLI)

AWS CLI Il s'agit d'un outil open source qui vous permet d'interagir avec les AWS services à l'aide de commandes dans votre interface de ligne de commande. Pour effectuer les étapes de cette section, vous devez disposer de la version 2 de l'AWS CLI.

Vous pouvez utiliser AWS CLI pour récupérer les journaux d'une invocation à l'aide de l'option de commande `--log-type`. La réponse inclut un champ `LogResult` qui contient jusqu'à 4 Ko de journaux codés en base64 provenant de l'invocation.

Example récupérer un ID de journal

L'exemple suivant montre comment récupérer un ID de journal à partir du champ `LogResult` d'une fonction nommée `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Vous devriez voir la sortie suivante:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example décoder les journaux

Dans la même invite de commandes, utilisez l'utilitaire `base64` pour décoder les journaux. L'exemple suivant montre comment récupérer les journaux encodés en base64 pour `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

Vous devriez voir la sortie suivante :

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

L'utilitaire `base64` est disponible sous Linux, macOS et Ubuntu sous Windows. Les utilisateurs de macOS auront peut-être besoin d'utiliser `base64 -D`.

Example Script get-logs.sh

Dans la même invite de commandes, utilisez le script suivant pour télécharger les cinq derniers événements de journalisation. Le script utilise `sed` pour supprimer les guillemets du fichier de sortie

et attend 15 secondes pour permettre la mise à disposition des journaux. La sortie comprend la réponse de Lambda, ainsi que la sortie de la commande `get-log-events`.

Copiez le contenu de l'exemple de code suivant et enregistrez-le dans votre répertoire de projet Lambda sous `get-logs.sh`.

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS et Linux (uniquement)

Dans la même invite de commandes, les utilisateurs macOS et Linux peuvent avoir besoin d'exécuter la commande suivante pour s'assurer que le script est exécutable.

```
chmod -R 755 get-logs.sh
```

Example récupérer les cinq derniers événements de journal

Dans la même invite de commande, exécutez le script suivant pour obtenir les cinq derniers événements de journalisation.

```
./get-logs.sh
```

Vous devriez voir la sortie suivante:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
```

```
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


Suppression de journaux

Les groupes de journaux ne sont pas supprimés automatiquement quand vous supprimez une fonction. Pour éviter de stocker des journaux indéfiniment, supprimez le groupe de journaux ou configurez une période de conservation à l'issue de laquelle les journaux sont supprimés automatiquement.

Utilisation de la bibliothèque Ruby logger

La bibliothèque Ruby logger renvoie des journaux simplifiés, faciles à lire. Utilisez l'utilitaire logger pour obtenir des informations détaillées, des messages et des codes d'erreur relatifs à votre fonction.

```
# lambda_function.rb

require 'logger'

def handler(event:, context:)
  logger = Logger.new($stdout)
  logger.info('## ENVIRONMENT VARIABLES')
  logger.info(ENV.to_a)
  logger.info('## EVENT')
  logger.info(event)
  event.to_a
end
```

La sortie de `logger` inclut le niveau de journal, l'horodatage et l'ID de demande.

```
START RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125 Version: $LATEST
[INFO]  2020-01-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ##
 ENVIRONMENT VARIABLES

[INFO]  2020-01-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125
    environ({'AWS_LAMBDA_LOG_GROUP_NAME': '/aws/lambda/my-function',
 'AWS_LAMBDA_LOG_STREAM_NAME': '2020/01/31/[$LATEST]1bbe51xmplb34a2788dbaa7433b0aa4d',
 'AWS_LAMBDA_FUNCTION_NAME': 'my-function', ...})

[INFO]  2020-01-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ## EVENT

[INFO]  2020-01-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    {'key':
 'value'}

END RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125
REPORT RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125  Duration: 2.75 ms   Billed
 Duration: 117 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 113.51 ms
XRAY TraceId: 1-5e34a66a-474xmpl7c2534a87870b4370   SegmentId: 073cxmpl3e442861
 Sampled: true
```

Instrumentation du code Ruby dans AWS Lambda

Lambda s'intègre avec AWS X-Ray pour vous permettre de suivre, de déboguer et d'optimiser les applications Lambda. Vous pouvez utiliser X-Ray pour suivre une demande lorsqu'elle qu'elle parcourt les ressources de votre application, de l'API frontale au stockage et à la base de données sur le backend. En ajoutant simplement la bibliothèque SDK X-Ray à votre configuration de build, vous pouvez enregistrer les erreurs et la latence pour tous les appels que votre fonction adresse à un service AWS.

Une fois que vous avez configuré le suivi actif, vous pouvez observer des demandes spécifiques via votre application. Le graphique de services X-Ray affiche des informations sur votre application et tous ses composants. L'exemple suivant montre une application dotée de deux fonctions. La fonction principale traite les événements et renvoie parfois des erreurs. La deuxième fonction en partant du haut traite les erreurs qui apparaissent dans le groupe de journaux du premier et utilise le kit SDK AWS pour appeler X-Ray, Amazon Simple Storage Service (Amazon S3) et Amazon CloudWatch logs.

Pour activer/désactiver le traçage actif sur votre fonction Lambda avec la console, procédez comme suit :

Pour activer le traçage actif

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Choisissez Configuration, puis choisissez Outils de surveillance et d'opérations.

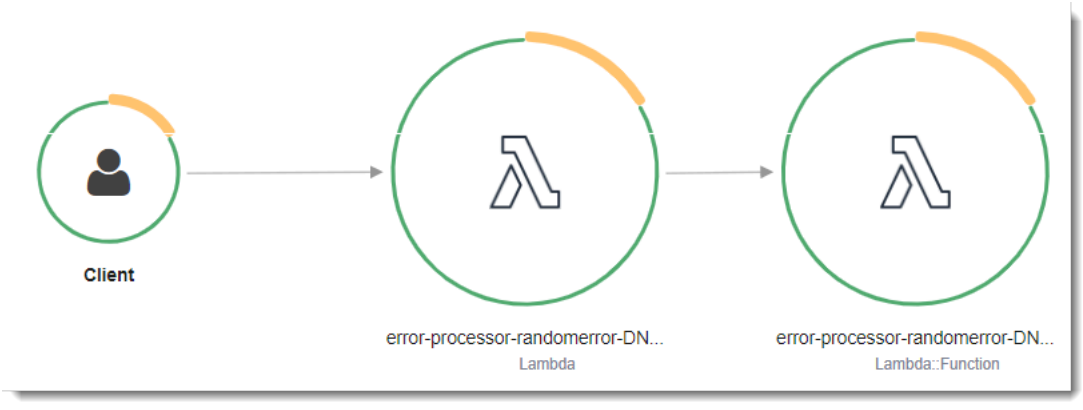
4. Dans Outils de surveillance supplémentaires, choisissez Modifier.
5. Sous CloudWatch Application Signals et AWS X-Ray sélectionnez Activer pour Suivis de service Lambda.
6. Choisissez Enregistrer.

> **Tarification**
>
> Vous pouvez utiliser gratuitement le traçage par X-Ray chaque mois jusqu'à une certaine limite dans le cadre de l'offre gratuite AWS. Au-delà de ce seuil, X-Ray facture le stockage et la récupération du suivi. Pour plus d'informations, consultez Tarification d'AWS X-Ray.

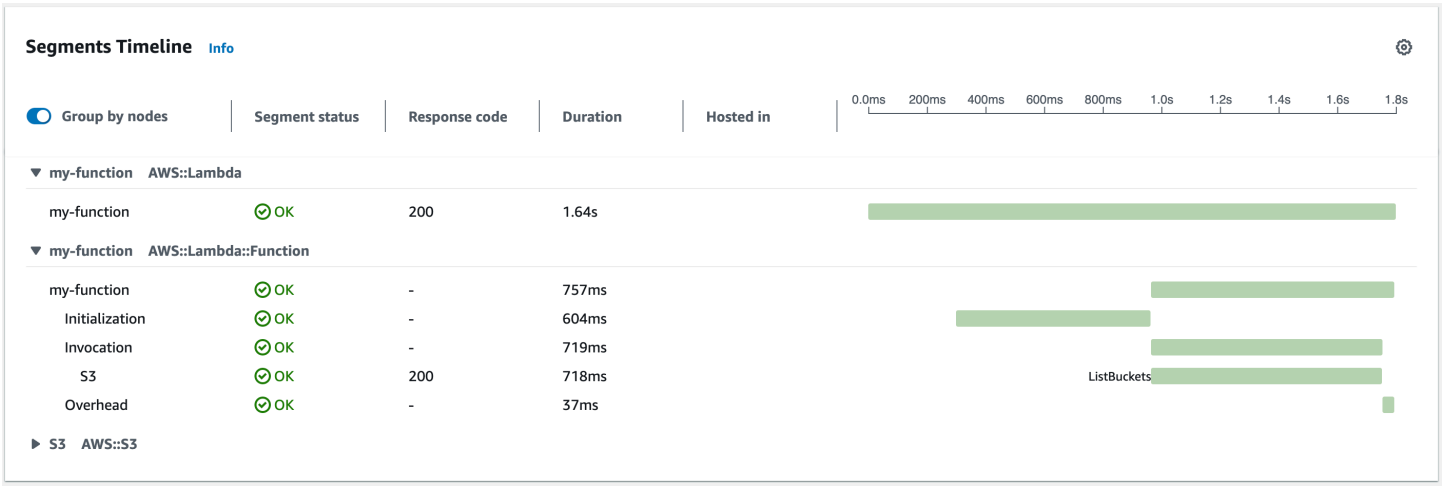
Votre fonction a besoin d'une autorisation pour charger des données de suivi vers X-Ray. Lorsque vous activez le suivi actif dans la console Lambda, Lambda ajoute les autorisations requises au rôle d'exécution de votre fonction. Sinon, ajoutez la politique AWSXRayDaemonWriteAccess au rôle d'exécution.

X-Ray ne trace pas toutes les requêtes vers votre application. X-Ray applique un algorithme d'échantillonnage pour s'assurer que le suivi est efficace, tout en fournissant un échantillon représentatif de toutes les demandes. Le taux d'échantillonnage est 1 demande par seconde et 5 % de demandes supplémentaires. Vous ne pouvez pas configurer ce taux d'échantillonnage X-Ray pour vos fonctions.

Dans X-Ray, un suivi enregistre des informations sur une demande traitée par un ou plusieurs services. Lambda enregistre deux segments par suivi, ce qui a pour effet de créer deux nœuds sur le graphique du service. L'image suivante met en évidence ces deux nœuds :



Le premier nœud sur la gauche représente le service Lambda qui reçoit la demande d'invocation. Le deuxième nœud représente votre fonction Lambda spécifique. L'exemple suivant illustre une trace avec ces deux segments. Les deux sont nommés my-function, mais l'un a pour origine `AWS::Lambda` et l'autre a pour origine `AWS::Lambda::Function`. Si le segment `AWS::Lambda` affiche une erreur, cela signifie que le service Lambda a rencontré un problème. Si le segment `AWS::Lambda::Function` affiche une erreur, cela signifie que votre fonction a rencontré un problème.



Cet exemple développe le segment `AWS::Lambda::Function` pour afficher ses trois sous-segments.

> **Note**
>
> AWS est en train de modifier le service Lambda. En raison de ces modifications, vous pouvez constater des différences mineures entre la structure et le contenu des messages du journal système et des segments de suivi émis par les différentes fonctions Lambda de votre Compte AWS.
>
> L'exemple de suivi présenté ici illustre le segment de fonction à l'ancienne. Les différences entre les segments à l'ancienne et de style moderne sont décrites dans les paragraphes suivants.
>
> Ces modifications seront mises en œuvre au cours des prochaines semaines, et toutes les fonctions dans toutes les Régions AWS, sauf les régions de Chine et GovCloud, passeront à l'utilisation du nouveau format des messages de journal et des segments de suivi.

Le segment de fonction à l'ancienne contient les sous-segments suivants :

- Initialization (Initialisation) : représente le temps passé à charger votre fonction et à exécuter le code d'initialisation. Ce sous-segment apparaît pour le premier événement traité par chaque instance de votre fonction.
- Invocation – Représente le temps passé à exécuter votre code de gestionnaire.
- Overhead (Travail supplémentaire) – Représente le temps que le fichier d'exécution Lambda passe à se préparer à gérer l'événement suivant.

Le segment de fonction de style moderne ne contient pas de sous-segment `Invocation`. À la place, les sous-segments du client sont directement rattachés au segment de fonction. Pour plus d'informations sur la structure des segments de fonction à l'ancienne et de style moderne, consultez the section called "Comprendre les suivis X-Ray".

Vous pouvez instrumenter le code de gestionnaire pour enregistrer les métadonnées et suivre les appels en aval. Pour enregistrer des détails sur les appels que le gestionnaire effectue vers d'autres ressources et services, utilisez le kit SDK X-Ray pour Ruby. Pour obtenir ce kit SDK, ajoutez le package `aws-xray-sdk` aux dépendances de votre application.

Example blank-ruby/function/Gemfile

```
# Gemfile
source 'https://rubygems.org'

gem 'aws-xray-sdk', '0.11.4'
gem 'aws-sdk-lambda', '1.39.0'
gem 'test-unit', '3.3.5'
```

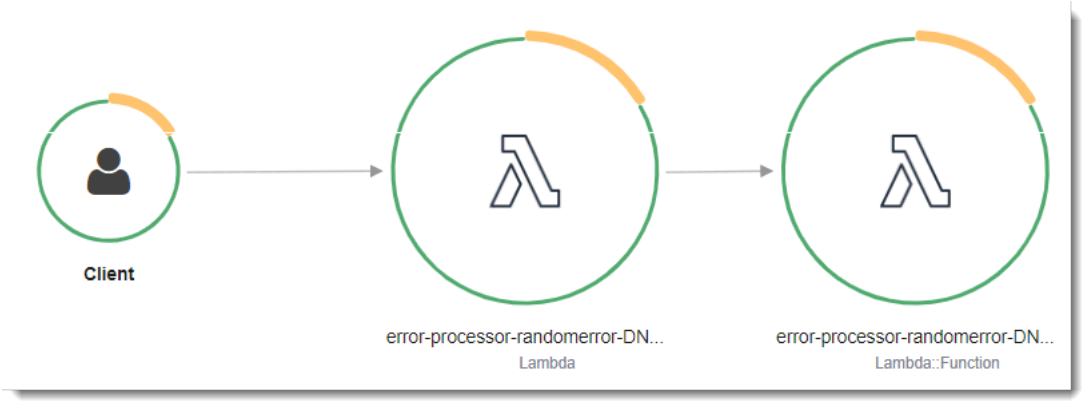
Pour instrumenter les clients AWS SDK, imposez le module `aws-xray-sdk/lambda` après avoir créé un client dans le code d'initialisation.

Example blank-ruby/function/lambda_function.rb – Suivi d'un client de kit SDK AWS

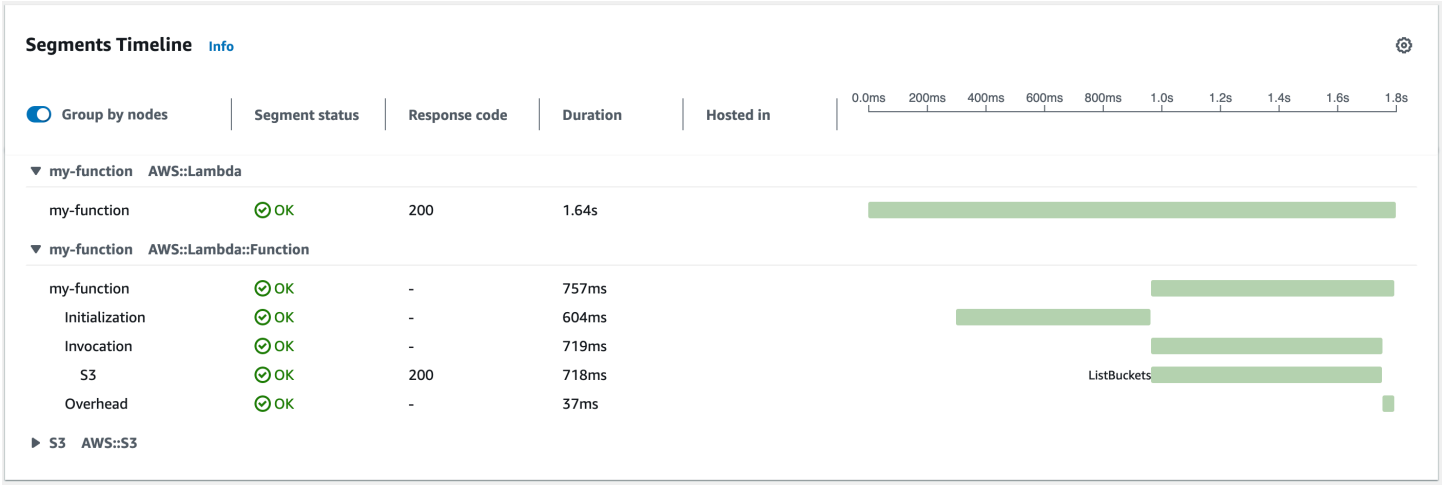
```
# lambda_function.rb
require 'logger'
require 'json'
require 'aws-sdk-lambda'
$client = Aws::Lambda::Client.new()
$client.get_account_settings()

require 'aws-xray-sdk/lambda'
```

```
def lambda_handler(event:, context:)
  logger = Logger.new($stdout)
  ...
```

Dans X-Ray, un suivi enregistre des informations sur une demande traitée par un ou plusieurs services. Lambda enregistre deux segments par suivi, ce qui a pour effet de créer deux nœuds sur le graphique du service. L'image suivante met en évidence ces deux nœuds :



Le premier nœud sur la gauche représente le service Lambda qui reçoit la demande d'invocation. Le deuxième nœud représente votre fonction Lambda spécifique. L'exemple suivant illustre une trace avec ces deux segments. Les deux sont nommés my-function, mais l'un a pour origine `AWS::Lambda` et l'autre a pour origine `AWS::Lambda::Function`. Si le segment `AWS::Lambda` affiche une erreur, cela signifie que le service Lambda a rencontré un problème. Si le segment `AWS::Lambda::Function` affiche une erreur, cela signifie que votre fonction a rencontré un problème.



Cet exemple développe le segment `AWS::Lambda::Function` pour afficher ses trois sous-segments.

> **Note**
>
> AWS est en train de modifier le service Lambda. En raison de ces modifications, vous pouvez constater des différences mineures entre la structure et le contenu des messages du journal système et des segments de suivi émis par les différentes fonctions Lambda de votre Compte AWS.
> L'exemple de suivi présenté ici illustre le segment de fonction à l'ancienne. Les différences entre les segments à l'ancienne et de style moderne sont décrites dans les paragraphes suivants.
> Ces modifications seront mises en œuvre au cours des prochaines semaines, et toutes les fonctions dans toutes les Régions AWS, sauf les régions de Chine et GovCloud, passeront à l'utilisation du nouveau format des messages de journal et des segments de suivi.

Le segment de fonction à l'ancienne contient les sous-segments suivants :

- Initialization (Initialisation) : représente le temps passé à charger votre fonction et à exécuter le code d'initialisation. Ce sous-segment apparaît pour le premier événement traité par chaque instance de votre fonction.
- Invocation – Représente le temps passé à exécuter votre code de gestionnaire.
- Overhead (Travail supplémentaire) – Représente le temps que le fichier d'exécution Lambda passe à se préparer à gérer l'événement suivant.

Le segment de fonction de style moderne ne contient pas de sous-segment `Invocation`. À la place, les sous-segments du client sont directement rattachés au segment de fonction. Pour plus d'informations sur la structure des segments de fonction à l'ancienne et de style moderne, consultez the section called "Comprendre les suivis X-Ray".

Vous pouvez également utiliser des clients HTTP, enregistrer des requêtes SQL et créer des sous-segments personnalisés avec des annotations et des métadonnées. Pour plus d'informations, consultez Kit SDK X-Ray pour Ruby dans le Guide du développeur AWS X-Ray.

Sections

- Activation du suivi actif avec l'API Lambda
- Activation du suivi actif avec CloudFormation
- Stockage des dépendances d'exécution dans une couche

Activation du suivi actif avec l'API Lambda

Pour gérer la configuration de suivi à l'aide de AWS CLI ou du kit AWS SDK, utilisez les opérations d'API suivantes :

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

L'exemple de commande AWS CLI suivant active le suivi actif sur une fonction nommée my-function.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Le mode de suivi fait partie de la configuration spécifique de la version lorsque vous publiez une version de votre fonction. Vous ne pouvez pas modifier le mode de suivi sur une version publiée.

Activation du suivi actif avec CloudFormation

Pour activer le suivi d'une ressource `AWS::Lambda::Function` dans un modèle CloudFormation, utilisez la propriété `TracingConfig`.

Example function-inline.yml – Configuration du suivi

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

Pour une ressource AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function`, utilisez la propriété `Tracing`.

Example template.yml – Configuration du suivi

```
Resources:
  function:
    Type: AWS::Serverless::Function
```

```
    Properties:
      Tracing: Active
      ...
```

Stockage des dépendances d'exécution dans une couche

Si vous utilisez le kit SDK X-Ray pour instrumenter des clients de kit SDK AWS avec votre code de fonction, votre package de déploiement peut devenir assez volumineux. Pour éviter de charger des dépendances d'environnement d'exécution chaque fois que vous mettez à jour votre code de fonction, empaquetez le kit SDK X-Ray dans une couche Lambda.

L'exemple suivant montre une ressource `AWS::Serverless::LayerVersion` qui stocke le kit SDK X-Ray pour Ruby.

Example template.yml : couche de dépendances

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: function/.
      Tracing: Active
      Layers:
        - !Ref libs
      ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-ruby-lib
      Description: Dependencies for the blank-ruby sample app.
      ContentUri: lib/.
      CompatibleRuntimes:
        - ruby2.5
```

Avec cette configuration, vous ne mettez à jour les fichiers de couche de bibliothèque que si vous modifiez vos dépendances d'exécution. Étant donné que le package de déploiement de la fonction contient uniquement votre code, cela peut contribuer à réduire les temps de chargement.

La création d'une couche de dépendances nécessite des modifications de génération pour créer l'archive des couches avant le déploiement. Pour un exemple fonctionnel, consultez l'exemple d'application blank-ruby.

Création de fonctions Lambda avec Java

Vous pouvez exécuter du code Java dans AWS Lambda. Lambda fournit des runtimes pour Java qui exécutent votre code afin de traiter des événements. Votre code s'exécute dans un environnement Amazon Linux qui inclut les AWS informations d'identification d'un rôle Gestion des identités et des accès AWS (IAM) que vous gérez.

Lambda prend en charge les environnements d’exécution Java suivants.

Name	Identifiant	Système d’exploitation	Date d’obsolescence	Créer la fonction de blocage	Mettre à jour la fonction de blocage
Java 25	`java25`	Amazon Linux 2	30 juin 2029	31 juillet 2029	31 août 2029
Java 21	`java21`	Amazon Linux 2	30 juin 2029	31 juillet 2029	31 août 2029
Java 17	`java17`	Amazon Linux 2	30 juin 2027	31 juillet 2027	31 août 2027
Java 11	`java11`	Amazon Linux 2	30 juin 2027	31 juillet 2027	31 août 2027
Java 8	`java8.al2`	Amazon Linux 2	30 juin 2027	31 juillet 2027	31 août 2027

AWS fournit les bibliothèques suivantes pour les fonctions Java. Ces bibliothèques sont disponibles via le Référentiel central Maven.

- com.amazonaws : aws-lambda-java-core (obligatoire) — Définit les interfaces des méthodes du gestionnaire et l'objet de contexte que le moteur d'exécution transmet au gestionnaire. Si vous définissez vos propres types d’entrée, c’est la seule bibliothèque dont vous avez besoin.
- com.amazonaws : aws-lambda-java-events — Types d'entrée pour les événements provenant de services qui invoquent des fonctions Lambda.
- com.amazonaws : aws-lambda-java-log 4j2 — Une bibliothèque d'ajout pour Apache Log4j 2 que vous pouvez utiliser pour ajouter l'ID de demande pour l'appel en cours dans vos journaux de fonctions.

- AWS SDK pour Java 2.0 — Le SDK AWS officiel pour le langage de programmation Java.

Ajoutez ces bibliothèques à votre définition de build comme suit :

Gradle

```
dependencies {
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.2'
    implementation 'com.amazonaws:aws-lambda-java-events:3.11.1'
    runtimeOnly 'com.amazonaws:aws-lambda-java-log4j2:1.5.1'
}
```

Maven

```
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-core</artifactId>
      <version>1.2.2</version>
    </dependency>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-events</artifactId>
      <version>3.11.1</version>
    </dependency>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-log4j2</artifactId>
      <version>1.5.1</version>
    </dependency>
  </dependencies>
```

Important

N'utilisez pas de composants privés de l'API JDK, tels que des champs privés, des méthodes ou des classes. Les composants d'API non publics peuvent être modifiés ou supprimés dans n'importe quelle mise à jour, ce qui peut entraîner l'interruption de votre application.

Pour créer une fonction Java

1. Ouvrez la console Lambda.
2. Sélectionnez Créer une fonction.
3. Configurez les paramètres suivants :
   - Nom de la fonction : saisissez le nom de la fonction.
   - Runtime : Choisissez Java 25.
4. Choisissez Créer une fonction.

La console crée une fonction Lambda avec une classe de gestionnaire nommée `Hello`. Étant donné que Java est un langage compilé, vous ne pouvez pas afficher ou modifier le code source dans la console Lambda, mais vous pouvez modifier sa configuration, l'invoquer et configurer des déclencheurs.

> **Note**
>
> Pour démarrer le développement d'applications dans votre environnement local, déployez l'un des exemples d'applications disponibles dans le GitHub référentiel de ce guide.

La classe `Hello` comporte une fonction nommée `handleRequest` qui accepte un objet d'événement et un objet de contexte. Il s'agit de la fonction de gestionnaire que Lambda appelle lors de l'invocation de la fonction. Le runtime de la fonction Java obtient les événements d'invocations à partir de Lambda et les transmet au gestionnaire. Dans la configuration de fonction, la valeur de gestionnaire est `example.Hello::handleRequest`.

Pour mettre à jour le code de la fonction, vous créez un package de déploiement, qui est une archive .zip contenant le code de votre fonction. Au fur et à mesure du développement de votre fonction, nous vous conseillons de stocker le code de votre fonction dans le contrôle de code source, d'ajouter des bibliothèques et d'automatiser les déploiements. Commencez par créer un package de déploiement et mettre à jour votre code dans la ligne de commande.

Le runtime de la fonction transmet un objet de contexte au gestionnaire, en plus de l'événement d'invocation. L'objet de contexte contient des informations supplémentaires sur l'invocation, la fonction et l'environnement d'exécution. Des informations supplémentaires sont disponibles dans les variables d'environnement.

Votre fonction Lambda est fournie avec un groupe de CloudWatch journaux Logs. La fonction runtime envoie les détails de chaque appel à CloudWatch Logs. Il relaie tous les journaux que votre fonction génère pendant l'invocation. Si votre fonction renvoie une erreur, Lambda met en forme l'erreur et la renvoie à l'appelant.

Rubriques

- Définition du gestionnaire de fonction Lambda dans Java
- Déployer des fonctions Lambda en Java avec des archives de fichiers .zip ou JAR
- Déployer des fonctions Lambda en Java avec des images conteneurs
- Utilisation de couches pour les fonctions Lambda Java
- Personnalisation de la sérialisation pour les fonctions Lambda Java
- Personnalisation du comportement de démarrage de l'environnement d'exécution Java pour les fonctions Lambda
- Utilisation de l'objet de contexte Lambda pour récupérer les informations de la fonction Java
- Journalisation et surveillance des fonctions Lambda Java
- Instrumentation du code Java dans AWS Lambda
- Exemples d'applications Java pour AWS Lambda

Définition du gestionnaire de fonction Lambda dans Java

Le gestionnaire de fonction Lambda est la méthode dans votre code de fonction qui traite les événements. Lorsque votre fonction est invoquée, Lambda exécute la méthode du gestionnaire. Votre fonction s'exécute jusqu'à ce que le gestionnaire renvoie une réponse, se ferme ou expire.

Cette page explique comment utiliser les gestionnaires de fonctions Lambda dans Java, notamment des options pour la configuration du projet, les conventions de dénomination et les pratiques exemplaires. Cette page inclut également un exemple de fonction Lambda Java qui collecte des informations relatives à une commande, produit un reçu sous forme de fichier texte et place ce fichier dans un compartiment Amazon Simple Storage Service (Amazon S3). Pour plus d'informations sur le déploiement de votre fonction après l'avoir écrite, consultez the section called "Déploiement d'archives de fichiers .zip" ou the section called "Déploiement d'images de conteneur".

Sections

- Configuration de votre projet de gestionnaire Java
- Exemple de code de fonction Lambda Java
- Définitions de classe valides pour les gestionnaires Java
- Convention de nommage du gestionnaire
- Définition et accès à l'objet d'événement d'entrée
- Accès et utilisation de l'objet de contexte Lambda
- Utilisation du SDK AWS pour Java v2 dans votre gestionnaire
- Accès aux variables d'environnement
- Utilisation de l'état global
- Pratiques exemplaires en matière de code pour les fonctions Lambda Java

Configuration de votre projet de gestionnaire Java

Lorsque vous utilisez des fonctions Lambda en Java, le processus consiste à écrire votre code, à le compiler et à déployer les artefacts compilés sur Lambda. Vous pouvez initialiser un projet Java Lambda de différentes manières. Par exemple, vous pouvez utiliser des outils comme l'archétype Maven pour les fonctions Lambda, la commande AWS SAM CLI sam init ou même une configuration de projet Java standard dans votre IDE préféré, comme IntelliJ IDEA ou Visual Studio Code. Vous pouvez également créer manuellement la structure de fichiers requise.

Un projet de fonction Lambda Java typique suit cette structure générale :

```
/project-root
    # src
        # main
            # java
                # example
                    # OrderHandler.java (contains main handler)
                    # <other_supporting_classes>
     # build.gradle OR pom.xml
```

Vous pouvez utiliser Maven ou Gradle pour créer votre projet et gérer les dépendances.

La logique de gestion principale de votre fonction réside dans un fichier Java situé sous le répertoire `src/main/java/example`. Dans l'exemple présenté sur cette page, nous nommons ce fichier `OrderHandler.java`. Outre ce fichier, vous pouvez inclure des classes Java supplémentaires selon vos besoins. Lorsque vous déployez votre fonction sur Lambda, assurez-vous de spécifier la classe Java qui contient la méthode de gestion principale que Lambda doit invoquer lors d'une invocation.

Exemple de code de fonction Lambda Java

L'exemple de code de fonction Lambda Java 21 suivant prend en compte les informations relatives à une commande, produit un reçu sous forme de fichier texte et place ce fichier dans un compartiment Amazon S3.

Example **`OrderHandler.java`**Fonction Lambda

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import software.amazon.awssdk.core.sync.RequestBody;
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.PutObjectRequest;
import software.amazon.awssdk.services.s3.model.S3Exception;

import java.nio.charset.StandardCharsets;

/**
 * Lambda handler for processing orders and storing receipts in S3.
 */
public class OrderHandler implements RequestHandler<OrderHandler.Order, String> {
```

```
    private static final S3Client S3_CLIENT = S3Client.builder().build();

    /**
     * Record to model the input event.
     */
    public record Order(String orderId, double amount, String item) {}

    @Override
    public String handleRequest(Order event, Context context) {
        try {
            // Access environment variables
            String bucketName = System.getenv("RECEIPT_BUCKET");
            if (bucketName == null || bucketName.isEmpty()) {
                throw new IllegalArgumentException("RECEIPT_BUCKET environment variable
is not set");
            }

            // Create the receipt content and key destination
            String receiptContent = String.format("OrderID: %s\nAmount: $%.2f\nItem:
%s",
                    event.orderId(), event.amount(), event.item());
            String key = "receipts/" + event.orderId() + ".txt";

            // Upload the receipt to S3
            uploadReceiptToS3(bucketName, key, receiptContent);

            context.getLogger().log("Successfully processed order " + event.orderId() +
                    " and stored receipt in S3 bucket " + bucketName);
            return "Success";

        } catch (Exception e) {
            context.getLogger().log("Failed to process order: " + e.getMessage());
            throw new RuntimeException(e);
        }
    }

    private void uploadReceiptToS3(String bucketName, String key, String
receiptContent) {
        try {
            PutObjectRequest putObjectRequest = PutObjectRequest.builder()
                    .bucket(bucketName)
                    .key(key)
                    .build();
```

```
            // Convert the receipt content to bytes and upload to S3
            S3_CLIENT.putObject(putObjectRequest,
 RequestBody.fromBytes(receiptContent.getBytes(StandardCharsets.UTF_8)));
        } catch (S3Exception e) {
            throw new RuntimeException("Failed to upload receipt to S3: " +
 e.awsErrorDetails().errorMessage(), e);
        }
    }
}
```

Ce fichier `OrderHandler.java` comprend les sections suivantes :

- `package example` : en Java, cela peut être n'importe quoi, mais cela doit correspondre à la structure de répertoires de votre projet. Ici, nous utilisons `package example`, car la structure de répertoires est `src/main/java/example`.
- Instructions `import` : utilisez ces instructions pour importer les classes Java que votre fonction Lambda nécessite.
- `public class OrderHandler ...` : définit votre classe Java et doit être une définition de classe valide.
- `private static final S3Client S3_CLIENT ...` : initialise un client S3 en dehors de l'une des méthodes de la classe. Cela oblige Lambda à exécuter ce code pendant la phase d'initialisation.
- `public record Order ...` : définissez la forme de l'événement d'entrée attendu dans cet enregistrement Java personnalisé.
- `public String handleRequest(Order event, Context context)` : il s'agit de la méthode de gestion principale, qui contient la logique principale de votre application.
- `private void uploadReceiptToS3(...) {}` : il s'agit d'une méthode auxiliaire référencée par la méthode de gestion principale `handleRequest`.

Exemples de fichiers build.gradle et pom.xml

Le fichier `build.gradle` ou `pom.xml` suivant accompagne cette fonction.

build.gradle

```
plugins {
    id 'java'
}
```

```
repositories {
    mavenCentral()
}

dependencies {
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.3'
    implementation 'software.amazon.awssdk:s3:2.28.29'
    implementation 'org.slf4j:slf4j-nop:2.0.16'
}

task buildZip(type: Zip) {
    from compileJava
    from processResources
    into('lib') {
        from configurations.runtimeClasspath
    }
}

java {
    sourceCompatibility = JavaVersion.VERSION_21
    targetCompatibility = JavaVersion.VERSION_21
}

build.dependsOn buildZip
```

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>example-java</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>example-java-function</name>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>21</maven.compiler.source>
        <maven.compiler.target>21</maven.compiler.target>
    </properties>
```

```
    <dependencies>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-core</artifactId>
            <version>1.2.3</version>
        </dependency>
        <dependency>
            <groupId>software.amazon.awssdk</groupId>
            <artifactId>s3</artifactId>
            <version>2.28.29</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-nop</artifactId>
            <version>2.0.16</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>3.5.2</version>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>3.4.1</version>
                <configuration>
                    <createDependencyReducedPom>false</createDependencyReducedPom>
                    <filters>
                        <filter>
                            <artifact>*:*</artifact>
                            <excludes>
                                <exclude>META-INF/*</exclude>
                                <exclude>META-INF/versions/**</exclude>
                            </excludes>
                        </filter>
                    </filters>
                </configuration>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
```

```
                            <goal>shade</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.13.0</version>
                <configuration>
                    <release>21</release>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Pour que cette fonction fonctionne correctement, son rôle d'exécution doit autoriser l'action `s3:PutObject`. Assurez-vous également de définir la variable d'environnement `RECEIPT_BUCKET`. Après une invocation réussie, le compartiment Amazon S3 doit contenir un fichier de reçu.

> **Note**
>
> Cette fonction peut nécessiter des paramètres de configuration supplémentaires pour s'exécuter correctement sans expiration de délai. Nous recommandons de configurer 256 Mo de mémoire et un délai d'attente de 10 secondes. La première invocation peut prendre plus de temps en raison d'un démarrage à froid. Les invocations suivantes devraient s'exécuter beaucoup plus rapidement en raison de la réutilisation de l'environnement d'exécution.

Définitions de classe valides pour les gestionnaires Java

Pour définir votre classe, la bibliothèque aws-lambda-java-core définit deux interfaces pour les méthodes du gestionnaire. Utilisez les interfaces fournies pour simplifier la configuration du gestionnaire et valider la signature de la méthode au moment de la compilation.

- com.amazonaws.services.lambda.runtime.RequestHandler
- com.amazonaws.services.lambda.runtime.RequestStreamHandler

L'interface `RequestHandler` est un type générique qui prend deux paramètres : le type d'entrée et le type de sortie. Les deux types doivent être des objets. Dans cet exemple, notre classe `OrderHandler` implémente `RequestHandler<OrderHandler.Order, String>`. Le type d'entrée est l'enregistrement `Order` que nous définissons dans la classe, et le type de sortie est `String`.

```
public class OrderHandler implements RequestHandler<OrderHandler.Order, String> {
    ...
}
```

Lorsque vous utilisez cette interface, le moteur d'exécution Java désérialise l'événement dans l'objet avec le type d'entrée et sérialise la sortie en texte. Utilisez cette interface lorsque la sérialisation intégrée fonctionne avec vos types d'entrée et de sortie.

Pour utiliser votre propre sérialisation, vous pouvez implémenter l'interface `RequestStreamHandler`. Avec cette interface, Lambda transmet à votre gestionnaire un flux d'entrée et un flux de sortie. Le gestionnaire lit les octets du flux d'entrée, écrit dans le flux de sortie et renvoie une valeur vide. Pour un exemple d'utilisation de l'environnement d'exécution Java 21, consultez HandlerStream.java.

Si vous travaillez uniquement avec des types de base et génériques (c'est-à-dire `String`, `Integer`, `List` ou `Map`) dans votre fonction Java, vous n'avez pas besoin d'implémenter d'interface. Par exemple, si votre fonction prend une entrée `Map<String, String>` et renvoie une valeur `String`, votre définition de classe et votre signature de gestionnaire peuvent ressembler à ce qui suit :

```
public class ExampleHandler {
    public String handleRequest(Map<String, String> input, Context context) {
        ...
    }
}
```

De plus, lorsque vous n'implémentez pas d'interface, l'objet de contexte est facultatif. Par exemple, votre définition de classe et votre signature de gestionnaire peuvent ressembler à ce qui suit :

```
public class NoContextHandler {
   public String handleRequest(Map<String, String> input) {
        ...
   }
}
```

Convention de nommage du gestionnaire

Pour les fonctions Lambda en Java, si vous implémentez l'interface `RequestHandler` ou `RequestStreamHandler`, la méthode de votre gestionnaire principal doit être nommée `handleRequest`. Incluez également la balise `@Override` au-dessus de votre méthode `handleRequest`. Lorsque vous déployez votre fonction sur Lambda, spécifiez le gestionnaire principal dans la configuration de votre fonction au format suivant :

- *`<package>`*.*`<Class>`* – Par exemple, `example.OrderHandler`.

Pour les fonctions Lambda en Java qui n'implémentent pas l'interface `RequestHandler` ou `RequestStreamHandler`, vous pouvez utiliser n'importe quel nom pour le gestionnaire. Lorsque vous déployez votre fonction sur Lambda, spécifiez le gestionnaire principal dans la configuration de votre fonction au format suivant :

- *`<package>`*.*`<Class>`*::*`<handler_method_name>`* – Par exemple, `example.Handler::mainHandler`.

Définition et accès à l'objet d'événement d'entrée

JSON est le format d'entrée le plus courant et standard pour les fonctions Lambda. Dans cet exemple, la fonction exige une entrée similaire à l'exemple suivant :

```
{
    "orderId": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

Lorsque vous utilisez des fonctions Lambda dans Java 17 ou version plus récente, vous pouvez définir la forme de l'événement d'entrée attendu sous la forme d'un enregistrement Java. Dans cet exemple, nous définissons un enregistrement au sein de la classe `OrderHandler` pour représenter un objet `Order` :

```
public record Order(String orderId, double amount, String item) {}
```

Cet enregistrement correspond à la forme d'entrée attendue. Après avoir défini votre enregistrement, vous pouvez écrire une signature de gestionnaire qui utilise une entrée JSON conforme à la définition

de l'enregistrement. L'environnement d'exécution Java désérialise automatiquement ce JSON en un objet Java. Vous pouvez ensuite accéder aux champs de l'objet. Par exemple, `event.orderId` récupère la valeur de `orderId` à partir de l'entrée d'origine.

> **Note**
>
> Les enregistrements Java sont une fonctionnalité des environnements d'exécution Java 17 et des versions plus récentes uniquement. Dans toutes les exécutions Java, vous pouvez utiliser une classe pour représenter les données des événements. Dans de tels cas, vous pouvez utiliser une bibliothèque comme jackson pour désérialiser les entrées JSON.

Autres types d'événements d'entrée

Il existe de nombreux événements d'entrée possibles pour les fonctions Lambda en Java :

- `Integer`, `Long`, `Double`, etc. – L'événement est un nombre sans mise en forme supplémentaire, par exemple, `3.5`. Le moteur d'exécution Java convertit la valeur en un objet du type spécifié.
- `String` – L'événement est une chaîne JSON incluant des guillemets, par exemple, `"My string"`. Le moteur d'exécution convertit la valeur en objet `String`, sans guillemets.
- `List<Integer>`, `List<String>`, `List<Object>`, etc. – L'événement est un tableau JSON. Le moteur d'exécution le désérialise en un objet du type ou de l'interface spécifié.
- `InputStream` – L'événement a un type JSON quelconque. Le moteur d'exécution transmet un flux d'octets du document au gestionnaire sans modification. Vous désérialisez l'entrée et écrivez la sortie dans un flux de sortie.
- Type de bibliothèque – Pour les événements envoyés par d'autres services AWS, utilisez les types de la bibliothèque aws-lambda-java-events. Par exemple, si votre fonction Lambda est invoquée par Amazon Simple Queue Service (SQS), utilisez l'objet `SQSEvent` comme entrée.

Accès et utilisation de l'objet de contexte Lambda

L'objet de contexte Lambda contient des informations sur l'invocation, la fonction et l'environnement d'exécution. Dans cet exemple, l'objet de contexte est de type `com.amazonaws.services.lambda.runtime.Context`, et constitue le deuxième argument de la fonction de gestion principale.

```
public String handleRequest(Order event, Context context) {
```

```
    ...
}
```

Si votre classe implémente l'interface RequestHandler ou RequestStreamHandler, l'objet de contexte est un argument obligatoire. Dans le cas contraire, l'objet de contexte est facultatif. Pour plus d'informations sur les signatures de gestionnaire acceptées valides, consultez the section called "Définitions de classe valides pour les gestionnaires Java".

Si vous appelez d'autres services à l'aide du kit SDK AWS, l'objet de contexte est requis dans quelques domaines clés. Par exemple, pour générer des journaux de fonction pour Amazon CloudWatch, vous pouvez utiliser la méthode `context.getLogger()` pour obtenir un objet `LambdaLogger` à enregistrer. Dans cet exemple, nous pouvons utiliser l'enregistreur pour enregistrer un message d'erreur si le traitement échoue pour une raison quelconque :

```
context.getLogger().log("Failed to process order: " + e.getMessage());
```

En dehors de la journalisation, vous pouvez également utiliser l'objet de contexte pour surveiller les fonctions. Pour plus d'informations sur la copie d'objets, consultez the section called "Contexte".

Utilisation du SDK AWS pour Java v2 dans votre gestionnaire

Vous utiliserez souvent les fonctions Lambda pour interagir avec d'autres ressources AWS ou pour les mettre à jour. Le moyen le plus simple d'interagir avec ces ressources est d'utiliser le SDK AWS pour Java v2.

> **Note**
>
> Le SDK AWS pour Java (v1) est en mode de maintenance et atteindra la fin du support le 31 décembre 2025. Nous vous recommandons de n'utiliser que le SDK AWS pour Java v2 à l'avenir.

Pour ajouter des dépendances du SDK à votre fonction, ajoutez-les dans votre fichier `build.gradle` pour Gradle ou dans votre fichier `pom.xml` pour Maven. Nous vous recommandons de n'ajouter que les bibliothèques dont vous avez besoin pour votre fonction. Dans l'exemple de code précédent, nous avons utilisé la bibliothèque `software.amazon.awssdk.services.s3`. Dans Gradle, vous pouvez ajouter cette dépendance en ajoutant la ligne suivante dans la section des dépendances de `build.gradle` :

```
implementation 'software.amazon.awssdk:s3:2.28.29'
```

Dans Maven, ajoutez les lignes suivantes dans la section `<dependencies>` de `pom.xml` :

```
    <dependency>
        <groupId>software.amazon.awssdk</groupId>
        <artifactId>s3</artifactId>
        <version>2.28.29</version>
    </dependency>
```

> **Note**
>
> Il ne s'agit peut-être pas de la version la plus récente du SDK. Choisissez la version du SDK adéquate pour votre application.

Importez ensuite les dépendances directement dans votre classe Java :

```
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.PutObjectRequest;
import software.amazon.awssdk.services.s3.model.S3Exception;
```

L'exemple de code initialise ensuite un client Amazon S3 comme suit :

```
private static final S3Client S3_CLIENT = S3Client.builder().build();
```

Dans cet exemple, nous avons initialisé notre client Amazon S3 en dehors de la fonction de gestionnaire principale pour éviter d'avoir à l'initialiser à chaque fois que nous invoquons notre fonction. Après avoir initialisé votre client SDK, vous pouvez l'utiliser pour interagir avec d'autres services AWS. L'exemple de code appelle l'API Amazon S3 `PutObject` comme suit :

```
PutObjectRequest putObjectRequest = PutObjectRequest.builder()
    .bucket(bucketName)
    .key(key)
    .build();

// Convert the receipt content to bytes and upload to S3
S3_CLIENT.putObject(putObjectRequest,
 RequestBody.fromBytes(receiptContent.getBytes(StandardCharsets.UTF_8)));
```

Accès aux variables d'environnement

Dans le code de votre gestionnaire, vous pouvez référencer n'importe quelle variable d'environnement à l'aide de la méthode `System.getenv()`. Dans cet exemple, nous référençons la variable d'environnement `RECEIPT_BUCKET` définie à l'aide de la ligne de code suivante :

```
String bucketName = System.getenv("RECEIPT_BUCKET");
if (bucketName == null || bucketName.isEmpty()) {
    throw new IllegalArgumentException("RECEIPT_BUCKET environment variable is not
 set");
}
```

Utilisation de l'état global

Lambda exécute votre code statique et le constructeur de classe pendant la phase d'initialisation avant d'invoquer votre fonction pour la première fois. Les ressources créées pendant l'initialisation restent en mémoire entre les invocations, ce qui vous évite d'avoir à les créer chaque fois que vous invoquez votre fonction.

Dans l'exemple de code, le code d'initialisation du client S3 se trouve en dehors de la méthode du gestionnaire principal. L'exécution initialise le client avant que la fonction n'exécute son premier événement, et le client reste disponible pour être réutilisé pendant toutes les invocations.

Pratiques exemplaires en matière de code pour les fonctions Lambda Java

Respectez les directives de la liste suivante pour utiliser les pratiques exemplaires de codage lors de la création de vos fonctions Lambda :

- Séparez le gestionnaire Lambda de votre logique principale. Cela vous permet de créer une fonction testable plus unitaire.
- Contrôlez les dépendances du package de déploiement de vos fonctions. L'environnement d'exécution AWS Lambda contient un certain nombre de bibliothèques. Pour activer le dernier ensemble de mises à jour des fonctionnalités et de la sécurité, Lambda met régulièrement à jour ces bibliothèques. Ces mises à jour peuvent introduire de subtiles modifications dans le comportement de votre fonction Lambda. Pour disposer du contrôle total des dépendances que votre fonction utilise, empaquetez toutes vos dépendances avec votre package de déploiement.
- Réduisez la complexité de vos dépendances. Privilégiez les infrastructures plus simples qui se chargent rapidement au démarrage de l'environnement d'exécution. Par exemple, préférez les

infrastructures d'injection (IoC) de dépendances Java plus simples comme Dagger ou Guice, à des plus complexes comme Spring Framework.

- Réduisez la taille de votre package de déploiement selon ses besoins relatifs à l'environnement d'exécution. Cela contribue à réduire le temps nécessaire au téléchargement et à la décompression de votre package de déploiement avant l'invocation. Pour les fonctions créées dans Java, évitez de charger la totalité de la bibliothèque du kit SDK AWS comme partie intégrante de votre package de déploiement. À la place, appuyez-vous de façon sélective sur les modules qui sélectionnent les composants du kit SDK dont vous avez besoin (par exemple, DynamoDB, modules du kit SDK Amazon S3 et bibliothèques principales Lambda).

Tirez parti de la réutilisation de l'environnement d'exécution pour améliorer les performances de votre fonction. Initialisez les clients SDK et les connexions à la base de données en dehors du gestionnaire de fonctions et mettez en cache les actifs statiques localement dans le répertoire `/tmp`. Les invocations ultérieures traitées par la même instance de votre fonction peuvent réutiliser ces ressources. Cela permet d'économiser des coûts, tout en réduisant le temps d'exécution de la fonction.

Pour éviter des éventuelles fuites de données entre les invocations, n'utilisez pas l'environnement d'exécution pour stocker des données utilisateur, des événements ou d'autres informations ayant un impact sur la sécurité. Si votre fonction repose sur un état réversible qui ne peut pas être stocké en mémoire dans le gestionnaire, envisagez de créer une fonction distincte ou des versions distinctes d'une fonction pour chaque utilisateur.

Utilisez une directive keep-alive pour maintenir les connexions persistantes. Lambda purge les connexions inactives au fil du temps. Si vous tentez de réutiliser une connexion inactive lorsque vous invoquez une fonction, cela entraîne une erreur de connexion. Pour maintenir votre connexion persistante, utilisez la directive Keep-alive associée à votre environnement d'exécution. Pour obtenir un exemple, consultez Réutilisation des connexions avec Keep-Alive dans Node.js.

Utilisez des variables d'environnement pour transmettre des paramètres opérationnels à votre fonction. Par exemple, si vous écrivez dans un compartiment Amazon S3 au lieu de coder en dur le nom du compartiment dans lequel vous écrivez, configurez le nom du compartiment comme variable d'environnement.

Évitez d'utiliser des invocations récursives dans votre fonction Lambda, lorsque la fonction s'invoque elle-même ou démarre un processus susceptible de l'invoquer à nouveau. Cela peut entraîner un volume involontaire d'invocations de fonction et des coûts accrus. Si vous constatez un volume

involontaire d'invocations, définissez immédiatement la simultanéité réservée à la fonction sur `0` afin de limiter toutes les invocations de la fonction, pendant que vous mettez à jour le code.

N'utilisez pas d'API non publiques non documentées dans le code de votre fonction Lambda. Pour les exécutions gérées AWS Lambda, Lambda applique périodiquement des mises à jour de sécurité et fonctionnelles aux API internes de Lambda. Ces mises à jour internes de l'API peuvent être incompatibles avec les versions antérieures, entraînant des conséquences imprévues telles que des échecs d'invocation si votre fonction dépend de ces API non publiques. Consultez la Référence d'API pour obtenir la liste des API accessibles au public.

Écriture du code idempotent. L'écriture de code idempotent pour vos fonctions garantit ne gestion identique des événements dupliqués. Votre code doit valider correctement les événements et gérer correctement les événements dupliqués. Pour de plus amples informations, veuillez consulterComment faire en sorte que ma fonction Lambda soit idempotente ?.

- Évitez d'utiliser le cache DNS de Java. Les fonctions Lambda mettent déjà en cache les réponses DNS. Si vous utilisez un autre cache DNS, vous risquez de subir des interruptions de connexion.

  La classe `java.util.logging.Logger` peut activer indirectement le cache DNS de la JVM. Pour remplacer les paramètres par défaut, définissez networkaddress.cache.ttl sur 0 avant de procéder à l'initialisation de `logger`. Exemple :

  ```
  public class MyHandler {
    // first set TTL property
    static{
     java.security.Security.setProperty("networkaddress.cache.ttl" , "0");
    }
   // then instantiate logger
    var logger = org.apache.logging.log4j.LogManager.getLogger(MyHandler.class);
  }
  ```

- Réduisez le temps que Lambda met à décompresser les packages de déploiement créés dans Java en plaçant vos fichiers `.jar` de dépendance dans un répertoire /lib distinct. Cette solution est plus rapide que le placement de tout le code de votre fonction dans un seul fichier .jar comprenant une multitude de fichiers `.class`. Pour obtenir des instructions, consultez Déployer des fonctions Lambda en Java avec des archives de fichiers .zip ou JAR.

Déployer des fonctions Lambda en Java avec des archives de fichiers .zip ou JAR

Le code de votre AWS Lambda fonction se compose de scripts ou de programmes compilés et de leurs dépendances. Pour déployer votre code de fonction vers Lambda, vous utilisez un package de déploiement. Lambda prend en charge deux types de packages de déploiement : les images conteneurs et les archives de fichiers .zip.

Cette page explique comment créer votre package de déploiement sous forme de fichier .zip ou Jar, puis utiliser le package de déploiement pour déployer votre code de fonction à AWS Lambda l'aide de AWS Command Line Interface (AWS CLI).

> **Important**
>
> Java 25 a introduit le support pour les caches Ahead-of-Time (AOT). Nous vous recommandons vivement de ne pas utiliser de caches AOT lors du déploiement de vos fonctions sous forme d'archives de fichiers .zip ou JAR, car les caches peuvent provoquer un comportement inattendu lorsque Lambda met à jour le runtime géré. Pour plus d'informations, voirAhead-of-Time (AOT) et caches CDS.

Sections

- Conditions préalables
- Outils et bibliothèques
- Création d’un package de déploiement avec Gradle
- Utilisation des couches pour les dépendances
- Création d’un package de déploiement avec Maven
- Chargement d’un package de déploiement avec la console Lambda
- Téléchargement d'un package de déploiement à l'aide du AWS CLI
- Téléchargement d'un package de déploiement avec AWS SAM

Conditions préalables

AWS CLI Il s'agit d'un outil open source qui vous permet d'interagir avec les AWS services à l'aide de commandes dans votre interface de ligne de commande. Pour effectuer les étapes de cette section, vous devez disposer de la version 2 de l'AWS CLI.

Outils et bibliothèques

AWS fournit les bibliothèques suivantes pour les fonctions Java. Ces bibliothèques sont disponibles via le Référentiel central Maven.

- com.amazonaws : aws-lambda-java-core (obligatoire) — Définit les interfaces des méthodes du gestionnaire et l'objet de contexte que le moteur d'exécution transmet au gestionnaire. Si vous définissez vos propres types d'entrée, c'est la seule bibliothèque dont vous avez besoin.
- com.amazonaws : aws-lambda-java-events — Types d'entrée pour les événements provenant de services qui invoquent des fonctions Lambda.
- com.amazonaws : aws-lambda-java-log 4j2 — Une bibliothèque d'ajout pour Apache Log4j 2 que vous pouvez utiliser pour ajouter l'ID de demande pour l'appel en cours dans vos journaux de fonctions.
- AWS SDK pour Java 2.0 — Le SDK AWS officiel pour le langage de programmation Java.

Ajoutez ces bibliothèques à votre définition de build comme suit :

Gradle

```
dependencies {
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.2'
    implementation 'com.amazonaws:aws-lambda-java-events:3.11.1'
    runtimeOnly 'com.amazonaws:aws-lambda-java-log4j2:1.5.1'
}
```

Maven

```
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-core</artifactId>
      <version>1.2.2</version>
    </dependency>
```

```
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-events</artifactId>
      <version>3.11.1</version>
    </dependency>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-log4j2</artifactId>
      <version>1.5.1</version>
    </dependency>
  </dependencies>
```

Afin de créer un package de déploiement, compilez votre code de fonction et vos dépendances dans un seul fichier .zip ou Java Archive (JAR). Pour Gradle, utilisez le type de build `Zip`. Pour Apache Maven, utilisez le plug-in Maven Shade. Pour télécharger votre package de déploiement, utilisez la console Lambda, l'API Lambda ou (). AWS Serverless Application Model AWS SAM

> **Note**
>
> Pour que la taille de votre package de déploiement reste petite, empaquetez les dépendances de votre fonction en couches. Les couches vous permettent de gérer vos dépendances de manière indépendante. Elles peuvent être utilisées par plusieurs fonctions et partagées avec d'autres comptes. Pour de plus amples informations, veuillez consulter Couches Lambda.

Création d'un package de déploiement avec Gradle

Afin de créer un package de déploiement avec le code et les dépendances de votre fonction dans Gradle, utilisez le type de build `Zip`. Voici un exemple tiré d'un fichier exemple build.gradle complet :

Example build.gradle – Tâche de génération

```
task buildZip(type: Zip) {
    into('lib') {
        from(jar)
        from(configurations.runtimeClasspath)
    }
}
```

Cette configuration de build donne lieu à un package de déploiement dans le répertoire `build/distributions`. Dans l'instruction `into('lib')`, la tâche `jar` assemble une archive jar contenant vos classes principales dans un dossier nommé `lib`. De plus, la tâche `configurations.runtimeClassPath` copie les bibliothèques de dépendances du chemin de classe de génération dans le même dossier `lib`.

Example build.gradle – Dépendances

```
dependencies {
    ...
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.2'
    implementation 'com.amazonaws:aws-lambda-java-events:3.11.1'
    implementation 'org.apache.logging.log4j:log4j-api:2.17.1'
    implementation 'org.apache.logging.log4j:log4j-core:2.17.1'
    runtimeOnly 'org.apache.logging.log4j:log4j-slf4j18-impl:2.17.1'
    runtimeOnly 'com.amazonaws:aws-lambda-java-log4j2:1.5.1'
    ...
}
```

Lambda charge les fichiers JAR dans l'ordre alphabétique Unicode. Si plusieurs fichiers JAR du répertoire `lib` contiennent la même classe, le premier fichier est utilisé. Vous pouvez utiliser le script shell suivant afin d'identifier les classes en double :

Example test-zip.sh

```
mkdir -p expanded
unzip path/to/my/function.zip -d expanded
find ./expanded/lib -name '*.jar' | xargs -n1 zipinfo -1 | grep '.*.class' | sort |
 uniq -c | sort
```

Utilisation des couches pour les dépendances

Vous pouvez empaqueter les dépendances de votre fonction par couches afin de réduire la taille de votre package de déploiement et de gérer les dépendances de manière indépendante. Pour de plus amples informations, veuillez consulter the section called "Layers".

Création d'un package de déploiement avec Maven

Pour créer un package de déploiement avec Maven, utilisez le plug-in Maven Shade. Le plug-in crée un fichier JAR qui contient le code de fonction compilé et toutes ses dépendances.

Example pom.xml – Configuration du plugin

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>3.2.2</version>
        <configuration>
          <createDependencyReducedPom>false</createDependencyReducedPom>
        </configuration>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
```

Pour créer le package de déploiement, utilisez la commande `mvn package`.

```
[INFO] Scanning for projects...
[INFO] -----------------------< com.example:java-maven >-----------------------
[INFO] Building java-maven-function 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
...
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ java-maven ---
[INFO] Building jar: target/java-maven-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-shade-plugin:3.2.2:shade (default) @ java-maven ---
[INFO] Including com.amazonaws:aws-lambda-java-core:jar:1.2.2 in the shaded jar.
[INFO] Including com.amazonaws:aws-lambda-java-events:jar:3.11.1 in the shaded jar.
[INFO] Including joda-time:joda-time:jar:2.6 in the shaded jar.
[INFO] Including com.google.code.gson:gson:jar:2.8.6 in the shaded jar.
[INFO] Replacing original artifact with shaded artifact.
[INFO] Replacing target/java-maven-1.0-SNAPSHOT.jar with target/java-maven-1.0-
SNAPSHOT-shaded.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  8.321 s
[INFO] Finished at: 2020-03-03T09:07:19Z
[INFO] ------------------------------------------------------------------------
```

Cette commande génère un fichier JAR dans le répertoire `target`.

> **Note**
>
> Si vous travaillez avec un JAR multi-version (MRJAR), vous devez inclure le MRJAR (c'est-à-dire le JAR ombré produit par le plugin Maven Shade) dans le répertoire `lib` et le zipper avant de charger votre package de déploiement sur Lambda. Sinon, Lambda pourrait ne pas décompresser correctement votre fichier JAR, ce qui ferait que votre fichier `MANIFEST.MF` serait ignoré.

Si vous utilisez la bibliothèque appender (`aws-lambda-java-log4j2`), vous devez également configurer un transformateur pour le plug-in Maven Shade. La bibliothèque de transformateurs combine les versions d'un fichier cache qui apparaissent à la fois dans la bibliothèque appender et dans Log4j.

Example pom.xml – Configuration du plugin avec l'appender Log4j 2

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>3.2.2</version>
        <configuration>
          <createDependencyReducedPom>false</createDependencyReducedPom>
        </configuration>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
            <configuration>
              <transformers>
                <transformer
implementation="com.github.edwgiz.maven_shade_plugin.log4j2_cache_transformer.PluginsCacheFile
                </transformer>
              </transformers>
            </configuration>
          </execution>
        </executions>
        <dependencies>
          <dependency>
```

```
            <groupId>com.github.edwgiz</groupId>
            <artifactId>maven-shade-plugin.log4j2-cachefile-transformer</artifactId>
            <version>2.13.0</version>
          </dependency>
        </dependencies>
      </plugin>
```

Chargement d'un package de déploiement avec la console Lambda

Pour créer une nouvelle fonction, vous devez d'abord créer la fonction dans la console, puis charger votre fichier .zip ou JAR. Pour mettre à jour une fonction existante, ouvrez la page de votre fonction, puis suivez la même procédure pour ajouter votre fichier .zip ou JAR mis à jour.

Si votre fichier package de déploiement fait moins de 50 Mo, vous pouvez créer ou mettre à jour une fonction en chargeant le fichier directement à partir de votre ordinateur local. Pour les fichiers .zip ou JAR de plus de 50 Mo, vous devez d'abord charger votre package dans un compartiment Amazon S3. Pour savoir comment charger un fichier dans un compartiment Amazon S3 à l'aide du AWS Management Console, consultez Getting started with Amazon S3. Pour télécharger des fichiers à l'aide de AWS CLI, voir Déplacer des objets dans le guide de AWS CLI l'utilisateur.

> **Note**
>
> Vous ne pouvez pas modifier le type de package de déploiement (.zip ou image de conteneur) d'une fonction existante. Par exemple, vous ne pouvez pas convertir une fonction d'image de conteneur pour utiliser un fichier d'archive .zip à la place. Vous devez créer une nouvelle fonction.

Pour créer une nouvelle fonction (console)

1. Ouvrez la page Fonctions de la console Lambda et choisissez Créer une fonction.
2. Choisissez Créer à partir de zéro.
3. Sous Informations de base, procédez comme suit :
    a. Pour Nom de la fonction, saisissez le nom de la fonction.
    b. Pour Exécution, sélectionnez l'exécution que vous souhaitez utiliser.
    c. (Facultatif) Pour Architecture, choisissez l'architecture de l'ensemble des instructions pour votre fonction. L'architecture par défaut est x86_64. Assurez-vous que le package

de déploiement .zip pour votre fonction est compatible avec l'architecture de l'ensemble d'instructions que vous sélectionnez.

4. (Facultatif) Sous Permissions (Autorisations), développez Change default execution role (Modifier le rôle d'exécution par défaut). Vous pouvez créer un rôle d'exécution ou en utiliser un existant.
5. Choisissez Créer une fonction. Lambda crée une fonction de base « Hello world » à l'aide de l'exécution de votre choix.

Pour charger une archive .zip ou JAR à partir de votre ordinateur local (console)

1. Sur la page Fonctions de la console Lambda, choisissez la fonction pour laquelle vous souhaitez charger le fichier .zip ou JAR.
2. Sélectionnez l'onglet Code.
3. Dans le volet Source du code, choisissez Charger à partir de.
4. Choisissez un fichier .zip ou .jar.
5. Pour charger un fichier .zip ou JAR, procédez comme suit :
   a. Sélectionnez Charger, puis choisissez votre fichier .zip ou JAR dans le sélecteur de fichiers.
   b. Choisissez Ouvrir.
   c. Choisissez Enregistrer.

Pour charger une archive .zip ou JAR depuis un compartiment Amazon S3 (console)

1. Sur la page Fonctions de la console Lambda, choisissez la fonction pour laquelle vous souhaitez charger un nouveau fichier .zip ou JAR.
2. Sélectionnez l'onglet Code.
3. Dans le volet Source du code, choisissez Charger à partir de.
4. Choisissez l'emplacement Amazon S3.
5. Collez l'URL du lien Amazon S3 de votre fichier .zip et choisissez Enregistrer.

Téléchargement d'un package de déploiement à l'aide du AWS CLI

Vous pouvez utiliser la AWS CLI pour créer une nouvelle fonction ou pour mettre à jour une fonction existante à l'aide d'un fichier .zip ou JAR. Utilisez la fonction de création et update-function-codeles

commandes pour déployer votre package .zip ou JAR. Si votre fichier est inférieur à 50 Mo, vous pouvez charger le package à partir d'un emplacement de fichier sur votre machine de génération locale. Pour les fichiers plus volumineux, vous devez charger votre package .zip ou JAR à partir d'un compartiment Amazon S3. Pour savoir comment charger un fichier dans un compartiment Amazon S3 à l'aide du AWS CLI, consultez la section Déplacer des objets dans le guide de AWS CLI l'utilisateur.

> **Note**
>
> Si vous chargez votre fichier .zip ou JAR depuis un compartiment Amazon S3 à l'aide du AWS CLI, le compartiment doit se trouver au même endroit Région AWS que votre fonction.

Pour créer une nouvelle fonction à l'aide d'un fichier .zip ou JAR avec le AWS CLI, vous devez spécifier les éléments suivants :

- Le nom de votre fonction (`--function-name`)
- L'exécution de votre fonction (`--runtime`)
- L'Amazon Resource Name (ARN) du rôle d'exécution de votre fonction (`--role`)
- Le nom de la méthode du gestionnaire dans votre code de fonction (`--handler`)

Vous devez également indiquer l'emplacement de votre fichier .zip ou JAR. Si votre fichier .zip ou JAR se trouve dans un dossier sur votre machine de génération locale, utilisez l'option `--zip-file` pour spécifier le chemin d'accès du fichier, comme le montre l'exemple de commande suivant.

```
aws lambda create-function --function-name myFunction \
--runtime java25 --handler example.handler \
--role arn:aws:iam::123456789012:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

Pour spécifier l'emplacement du fichier .zip dans un compartiment Amazon S3, utilisez l'option `--code` comme le montre l'exemple de commande suivant. Vous devez uniquement utiliser le paramètre `S3ObjectVersion` pour les objets soumis à la gestion des versions.

```
aws lambda create-function --function-name myFunction \
--runtime java25 --handler example.handler \
--role arn:aws:iam::123456789012:role/service-role/my-lambda-role \
```

```
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

Pour mettre à jour une fonction existante à l'aide de l'interface de ligne de commande, vous devez spécifier le nom de votre fonction à l'aide du paramètre `--function-name`. Vous devez également spécifier l'emplacement du fichier .zip que vous souhaitez utiliser pour mettre à jour votre code de fonction. Si votre fichier .zip se trouve dans un dossier sur votre machine de génération locale, utilisez l'option `--zip-file` pour spécifier le chemin d'accès du fichier, comme le montre l'exemple de commande suivant.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Pour spécifier l'emplacement du fichier .zip dans un compartiment Amazon S3, utilisez les options `--s3-bucket` et `--s3-key` comme le montre l'exemple de commande suivant. Vous devez uniquement utiliser le paramètre `--s3-object-version` pour les objets soumis à la gestion des versions.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Téléchargement d'un package de déploiement avec AWS SAM

Vous pouvez l'utiliser AWS SAM pour automatiser les déploiements de votre code de fonction, de votre configuration et de vos dépendances. AWS SAM est une extension de CloudFormation qui fournit une syntaxe simplifiée pour définir des applications sans serveur. L'exemple de modèle suivant définit une fonction avec un package de déploiement dans le répertoire `build/distributions` utilisé par Gradle :

Example template.yml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: An AWS Lambda application that calls the Lambda API.
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
```

```
      CodeUri: build/distributions/java-basic.zip
      Handler: example.Handler
      Runtime: java25
      Description: Java function
      MemorySize: 512
      Timeout: 10
      # Function's execution role
      Policies:
        - AWSLambdaBasicExecutionRole
        - AWSLambda_ReadOnlyAccess
        - AWSXrayWriteOnlyAccess
        - AWSLambdaVPCAccessExecutionRole
      Tracing: Active
```

Pour créer la fonction, utilisez les commandes `package` et `deploy`. Ces commandes sont des personnalisations de l AWS CLI. Elles encapsulent d'autres commandes pour charger le package de déploiement sur Amazon S3, réécrivent le modèle avec l'URI de l'objet, et mettent à jour le code de la fonction.

L'exemple de script suivant exécute une build Gradle et télécharge le package de déploiement qu'il crée. Il crée une CloudFormation pile la première fois que vous l'exécutez. Si la pile existe déjà, le script la met à jour.

Example deploy.sh

```
#!/bin/bash
set -eo pipefail
aws cloudformation package --template-file template.yml --s3-bucket MY_BUCKET --output-
template-file out.yml
aws cloudformation deploy --template-file out.yml --stack-name java-basic --
capabilities CAPABILITY_NAMED_IAM
```

Afin d'obtenir un exemple de travail complet, consultez les exemples d'applications suivants :

Exemples d'applications Lambda en Java

- example-java : fonction Java qui montre comment utiliser Lambda pour traiter les commandes. Cette fonction montre comment définir et désérialiser un objet d'événement d'entrée personnalisé, utiliser le AWS SDK et enregistrer les sorties.
- java-basic – Ensemble de fonctions Java minimales avec des tests unitaires et une configuration de journalisation variable.

- java events – Ensemble de fonctions Java contenant du code squelette permettant de gérer les événements de divers services tels qu'Amazon API Gateway, Amazon SQS et Amazon Kinesis. Ces fonctions utilisent la dernière version de la aws-lambda-java-eventsbibliothèque (3.0.0 et versions ultérieures). Ces exemples ne nécessitent pas le AWS SDK comme dépendance.
- s3-java – Fonction Java qui traite les événements de notification d'Amazon S3 et utilise la bibliothèque de classes Java (JCL) pour créer des miniatures à partir de fichiers d'image chargés.
- layer-java : fonction Java qui illustre comment utiliser une couche Lambda pour empaqueter les dépendances séparément du code de votre fonction principale.

Déployer des fonctions Lambda en Java avec des images conteneurs

Il existe trois méthodes pour créer une image de conteneur pour une fonction Lambda Java :

- Utilisation d'une image AWS de base pour Java

  Les images de base AWS sont préchargées avec une exécution du langage, un client d'interface d'exécution pour gérer l'interaction entre Lambda et votre code de fonction, et un émulateur d'interface d'exécution pour les tests locaux.

- Utilisation d'une image de base AWS uniquement pour le système d'exploitation

  AWS Les images de base réservées au système d'exploitation contiennent une distribution Amazon Linux et l'émulateur d'interface d'exécution. Ces images sont couramment utilisées pour créer des images de conteneur pour les langages compilés, tels que Go et Rust, et pour une langue ou une version linguistique pour laquelle Lambda ne fournit pas d'image de base, comme Node.js 19. Vous pouvez également utiliser des images de base uniquement pour le système d'exploitation pour implémenter un environnement d'exécution personnalisé. Pour rendre l'image compatible avec Lambda, vous devez inclure le client d'interface d'exécution pour Java dans l'image.

- Utilisation d'une image non AWS basique

  Vous pouvez utiliser une autre image de base à partir d'un autre registre de conteneur, comme Alpine Linux ou Debian. Vous pouvez également utiliser une image personnalisée créée par votre organisation. Pour rendre l'image compatible avec Lambda, vous devez inclure le client d'interface d'exécution pour Java dans l'image.

> **Tip**
>
> Pour réduire le temps nécessaire à l'activation des fonctions du conteneur Lambda, consultez Utiliser des générations en plusieurs étapes (français non garanti) dans la documentation Docker. Pour créer des images de conteneur efficaces, suivez la section Bonnes pratiques pour l'écriture de Dockerfiles (français non garanti).

Cette page explique comment créer, tester et déployer des images de conteneur pour Lambda.

Rubriques

- AWS images de base pour Java
- Utilisation d'une image AWS de base pour Java
- Utilisation d’une autre image de base avec le client d’interface d’exécution

AWS images de base pour Java

AWS fournit les images de base suivantes pour Java :

Étiquettes	Environnement d’exécution	Système d’exploitation	Dockerfile	Obsolescence
25	Java 25	Amazon Linux	Dockerfile pour Java 2.5 sur GitHub	30 juin 2029
21	Java 21	Amazon Linux	Dockerfile pour Java 21 sur GitHub	30 juin 2029
17	Java 17	Amazon Linux 2	Dockerfile pour Java 17 sur GitHub	30 juin 2027
11	Java 11	Amazon Linux 2	Dockerfile pour Java 11 sur GitHub	30 juin 2027
8.al2	Java 8	Amazon Linux 2	Dockerfile pour Java 8 sur GitHub	30 juin 2027

Référentiel Amazon ECR : gallery.ecr. aws/lambda/java

Les images de base de Java 21 et versions ultérieures sont basées sur l’image de conteneur minimale Amazon Linux 2023. Les images de base antérieures utilisaient Amazon Linux 2. AL2Le 023 offre plusieurs avantages par rapport à Amazon Linux 2, notamment un encombrement de déploiement réduit et des versions mises à jour de bibliothèques telles que `glibc`.

AL2Les images basées sur le format 023 utilisent `microdnf` (lien symbolique sous forme `dnf`) comme gestionnaire de packages au lieu de `yum`, qui est le gestionnaire de packages par défaut

dans Amazon Linux 2. `microdnf`est une implémentation autonome de. `dnf` Pour obtenir la liste des packages inclus dans les images AL2 basées sur la version 023, reportez-vous aux colonnes Conteneur minimal de la section Comparaison des packages installés sur les images de conteneurs Amazon Linux 2023. Pour plus d'informations sur les différences entre AL2 023 et Amazon Linux 2, consultez Présentation du runtime Amazon Linux 2023 AWS Lambda sur le blog AWS Compute.

> **Note**
>
> Pour exécuter des images AL2 basées sur 023 localement, y compris avec AWS Serverless Application Model (AWS SAM), vous devez utiliser Docker version 20.10.10 ou ultérieure.

Utilisation d'une image AWS de base pour Java

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- Java (par exemple, Amazon Corretto)
- Apache Maven ou Gradle
- AWS CLI version 2
- Docker (version minimale 25.0.0)
- Le plug-in buildx Docker.

Création d'une image à partir d'une image de base

Maven

1. Exécutez la commande suivante pour créer un projet Maven en utilisant l'archétype de Lambda. Les paramètres suivants sont obligatoires :

   - service — Le Service AWS client à utiliser dans la fonction Lambda. Pour une liste des sources disponibles, voir aws-sdk-java-v2/services on GitHub.
   - region — L' Région AWS endroit où vous souhaitez créer la fonction Lambda.
   - groupId : l'espace de noms complet du package de votre application.
   - artifactId : le nom de votre projet. Cette valeur devient le nom du répertoire de votre projet.

Sous Linux et macOS, exécutez cette commande :

```
mvn -B archetype:generate \
   -DarchetypeGroupId=software.amazon.awssdk \
   -DarchetypeArtifactId=archetype-lambda -Dservice=s3 -Dregion=US_WEST_2 \
   -DgroupId=com.example.myapp \
   -DartifactId=myapp
```

Dans PowerShell, exécutez cette commande :

```
mvn -B archetype:generate `
   "-DarchetypeGroupId=software.amazon.awssdk" `
   "-DarchetypeArtifactId=archetype-lambda" "-Dservice=s3" "-Dregion=US_WEST_2"
 `
   "-DgroupId=com.example.myapp" `
   "-DartifactId=myapp"
```

L'archétype Maven pour Lambda est préconfiguré pour compiler avec Java SE 8 et inclut une dépendance au kit AWS SDK pour Java. Si vous créez votre projet avec un archétype différent ou en utilisant une autre méthode, vous devez configurer le compilateur Java pour Maven et déclarer le kit SDK comme une dépendance.

2. Ouvrez le répertoire *myapp*`/src/main/java/com/example/`*myapp* et trouvez le fichier `App.java`. Il s'agit du code de la fonction Lambda. Vous pouvez utiliser l'exemple de code fourni pour les tests ou le remplacer par le vôtre.
3. Retournez au répertoire racine du projet, puis créez un nouveau Dockerfile avec la configuration suivante :

   - Définir la propriété `FROM` sur l'URI de l'image de base.
   - Définir l'argument `CMD` pour le gestionnaire de la fonction Lambda.

   Notez que l'exemple de Dockerfile n'inclut pas d'instruction USER. Lorsque vous déployez une image de conteneur sur Lambda, Lambda définit automatiquement un utilisateur Linux par défaut disposant d'autorisations de moindre privilège. Ceci est différent du comportement standard de Docker qui est défini par défaut par l'utilisateur `root` lorsqu'aucune instruction USER n'est fournie.

Example Dockerfile

```
FROM public.ecr.aws/lambda/java:21

# Copy function code and runtime dependencies from Maven layout
COPY target/classes ${LAMBDA_TASK_ROOT}
COPY target/dependency/* ${LAMBDA_TASK_ROOT}/lib/

# Set the CMD to your handler (could also be done as a parameter override
 outside of the Dockerfile)
CMD [ "com.example.myapp.App::handleRequest" ]
```

4. Compilez le projet et rassemblez les dépendances de l'environnement d'exécution.

```
mvn compile dependency:copy-dependencies -DincludeScope=runtime
```

5. Générez l'image Docker à l'aide de la commande docker build. L'exemple suivant nomme l'image `docker-image` et lui donne la balise `test`. Pour rendre votre image compatible avec Lambda, vous devez utiliser l'option `--provenance=false`.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-
image:test .
```

> **Note**
>
> La commande spécifie l'option `--platform linux/amd64` pour garantir la compatibilité de votre conteneur avec l'environnement d'exécution Lambda, quelle que soit l'architecture de votre machine de génération. Si vous avez l'intention de créer une fonction Lambda à l'aide de l'architecture du jeu ARM64 d'instructions, veillez à modifier la commande pour utiliser l'`--platform linux/arm64`option à la place.

Gradle

1. Créez un répertoire pour le projet, puis passez à ce répertoire.

```
mkdir example
```

```
cd example
```

2. Exécutez la commande suivante pour que Gradle génère un nouveau projet d'application Java dans le répertoire `example` de votre environnement. Pour Sélectionner le script de construction DSL, choisissez 2 : Groovy.

```
gradle init --type java-application
```

3. Ouvrez le répertoire */example*`/app/src/main/java/`*example* et trouvez le fichier `App.java`. Il s'agit du code de la fonction Lambda. Vous pouvez utiliser l'exemple de code suivant pour le tester, ou le remplacer par le vôtre.

   Example App.java

```
package com.example;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
public class App implements RequestHandler<Object, String> {
    public String handleRequest(Object input, Context context) {
        return "Hello world!";
    }
}
```

4. Ouvrez le fichier `build.gradle`. Si vous utilisez l'exemple de code de fonction de l'étape précédente, remplacez le contenu de `build.gradle` par ce qui suit. Si vous utilisez votre propre code de fonction, modifiez votre fichier `build.gradle` selon vos besoins.

   Example build.gradle (Groovy DSL)

```
plugins {
  id 'java'
}
group 'com.example'
version '1.0-SNAPSHOT'
sourceCompatibility = 1.8
repositories {
  mavenCentral()
}
dependencies {
  implementation 'com.amazonaws:aws-lambda-java-core:1.2.1'
}
jar {
```

```
  manifest {
      attributes 'Main-Class': 'com.example.App'
  }
}
```

5. La commande `gradle init` de l'étape 2 a également généré un cas de test fictif dans le répertoire `app/test`. Pour les besoins de ce tutoriel, passez outre l'exécution des tests en supprimant le répertoire `/test`.
6. Générez le projet.

```
gradle build
```

7. Dans le répertoire racine du projet (`/example`), créez un fichier Docker avec la configuration suivante :

   - Définir la propriété `FROM` sur l'URI de l'image de base.
   - Utilisez la commande COPY pour copier le code de la fonction et les dépendances de l'environnement d'exécution dans `{LAMBDA_TASK_ROOT}`, une variable d'environnement définie par Lambda.
   - Définir l'argument `CMD` pour le gestionnaire de la fonction Lambda.

   Notez que l'exemple de Dockerfile n'inclut pas d'instruction USER. Lorsque vous déployez une image de conteneur sur Lambda, Lambda définit automatiquement un utilisateur Linux par défaut disposant d'autorisations de moindre privilège. Ceci est différent du comportement standard de Docker qui est défini par défaut par l'utilisateur `root` lorsqu'aucune instruction USER n'est fournie.

   Example Dockerfile

```
FROM public.ecr.aws/lambda/java:21

# Copy function code and runtime dependencies from Gradle layout
COPY app/build/classes/java/main ${LAMBDA_TASK_ROOT}

# Set the CMD to your handler (could also be done as a parameter override
 outside of the Dockerfile)
CMD [ "com.example.App::handleRequest" ]
```

8. Générez l'image Docker à l'aide de la commande docker build. L'exemple suivant nomme l'image `docker-image` et lui donne la balise `test`. Pour rendre votre image compatible avec Lambda, vous devez utiliser l'option `--provenance=false`.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test .
```

> **Note**
>
> La commande spécifie l'option `--platform linux/amd64` pour garantir la compatibilité de votre conteneur avec l'environnement d'exécution Lambda, quelle que soit l'architecture de votre machine de génération. Si vous avez l'intention de créer une fonction Lambda à l'aide de l'architecture du jeu ARM64 d'instructions, veillez à modifier la commande pour utiliser l'`--platform linux/arm64`option à la place.

(Facultatif) Testez l'image localement

1. Démarrez votre image Docker à l'aide de la commande docker run. Dans cet exemple, `docker-image` est le nom de l'image et `test` est la balise.

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

Cette commande exécute l'image en tant que conteneur et crée un point de terminaison local à `localhost:9000/2015-03-31/functions/function/invocations`.

> **Note**
>
> Si vous avez créé l'image Docker pour l'architecture du jeu ARM64 d'instructions, veillez à utiliser l'`--platform linux/arm64`option au lieu de`--platform linux/amd64`.

2. À partir d'une nouvelle fenêtre de terminal, publiez un événement au point de terminaison local.

Linux/macOS

Sous Linux et macOS, exécutez la commande `curl` suivante :

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

Cette commande invoque la fonction avec un événement vide et renvoie une réponse. Si vous utilisez votre propre code de fonction plutôt que l'exemple de code de fonction, vous pouvez invoquer la fonction avec une charge utile JSON. Exemple :

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

Dans PowerShell, exécutez la `Invoke-WebRequest` commande suivante :

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Cette commande invoque la fonction avec un événement vide et renvoie une réponse. Si vous utilisez votre propre code de fonction plutôt que l'exemple de code de fonction, vous pouvez invoquer la fonction avec une charge utile JSON. Exemple :

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. Obtenez l'ID du conteneur.

```
docker ps
```

4. Utilisez la commande docker kill pour arrêter le conteneur. Dans cette commande, remplacez `3766c4ab331c` par l'ID du conteneur de l'étape précédente.

```
docker kill 3766c4ab331c
```

Déploiement de l'image

Pour charger l'image sur Amazon RIE et créer la fonction Lambda

1. Exécutez la get-login-passwordcommande pour authentifier la CLI Docker auprès de votre registre Amazon ECR.

   - Définissez la `--region` valeur à l' Région AWS endroit où vous souhaitez créer le référentiel Amazon ECR.
   - `111122223333`Remplacez-le par votre Compte AWS identifiant.

   ```
   aws ecr get-login-password --region us-east-1 | docker login --username AWS --
   password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
   ```

2. Créez un référentiel dans Amazon ECR à l'aide de la commande create-repository.

   ```
   aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
   scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
   ```

   > **Note**
   >
   > Le référentiel Amazon ECR doit être Région AWS identique à la fonction Lambda.

   En cas de succès, vous obtenez une réponse comme celle-ci :

   ```
   {
       "repository": {
           "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
   world",
           "registryId": "111122223333",
           "repositoryName": "hello-world",
           "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
   world",
           "createdAt": "2023-03-09T10:39:01+00:00",
           "imageTagMutability": "MUTABLE",
           "imageScanningConfiguration": {
               "scanOnPush": true
           },
           "encryptionConfiguration": {
   ```

```
            "encryptionType": "AES256"
        }
    }
}
```

3. Copiez le `repositoryUri` à partir de la sortie de l'étape précédente.
4. Exécutez la commande docker tag pour étiqueter votre image locale dans votre référentiel Amazon ECR en tant que dernière version. Dans cette commande :
   - `docker-image:test` est le nom et la balise de votre image Docker. Il s'agit du nom et de la balise de l'image que vous avez spécifiés dans la commande `docker build`.
   - Remplacez <ECRrepositoryUri> par l'`repositoryUri` que vous avez copié. Assurez-vous d'inclure `:latest` à la fin de l'URI.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

   Exemple :

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Exécutez la commande docker push pour déployer votre image locale dans le référentiel Amazon ECR. Assurez-vous d'inclure `:latest` à la fin de l'URI du référentiel.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Créez un rôle d'exécution pour la fonction, si vous n'en avez pas déjà un. Vous aurez besoin de l'Amazon Resource Name (ARN) du rôle à l'étape suivante.
7. Créez la fonction Lambda. Pour `ImageUri`, indiquez l'URI du référentiel mentionné précédemment. Assurez-vous d'inclure `:latest` à la fin de l'URI.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> Vous pouvez créer une fonction en utilisant une image d'un autre AWS compte, à condition que l'image se trouve dans la même région que la fonction Lambda. Pour de plus amples informations, veuillez consulter Autorisations entre comptes Amazon ECR.

8. Invoquer la fonction.

```
aws lambda invoke --function-name hello-world response.json
```

Vous devriez obtenir une réponse comme celle-ci :

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Pour voir la sortie de la fonction, consultez le fichier `response.json`.

Pour mettre à jour le code de fonction, vous devez créer à nouveau l'image, télécharger la nouvelle image dans le référentiel Amazon ECR, puis utiliser la update-function-codecommande pour déployer l'image sur la fonction Lambda.

Lambda résout l'étiquette d'image en hachage d'image spécifique. Cela signifie que si vous pointez la balise d'image qui a été utilisée pour déployer la fonction vers une nouvelle image dans Amazon ECR, Lambda ne met pas automatiquement à jour la fonction pour utiliser la nouvelle image.

Pour déployer la nouvelle image sur la même fonction Lambda, vous devez utiliser la update-function-codecommande, même si la balise d'image dans Amazon ECR reste la même. Dans l'exemple suivant, l'option `--publish` crée une version de la fonction à l'aide de l'image du conteneur mise à jour.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Utilisation d’une autre image de base avec le client d’interface d’exécution

Si vous utilisez une image de base uniquement pour le système d’exploitation ou une autre image de base, vous devez inclure le client d’interface d’exécution dans votre image. Le client d’interface d’exécution étend leAPI de runtime, qui gère l’interaction entre Lambda et votre code de fonction.

Installez le client d’interface d’exécution pour Java dans votre Dockerfile ou en tant que dépendance dans votre projet. Par exemple, pour installer le client d’interface d’exécution à l’aide du gestionnaire de packages Maven, ajoutez ce qui suit à votre fichier `pom.xml` :

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-lambda-java-runtime-interface-client</artifactId>
    <version>2.3.2</version>
</dependency>
```

Pour plus de détails sur le package, consultez Client d’interface d’exécution AWS Lambda dans le référentiel central Maven. Vous pouvez également consulter le code source du client de l'interface d'exécution dans le GitHub référentiel AWS Lambda Java Support Libraries.

L’exemple suivant montre comment créer une image de conteneur pour Java à l’aide d’une image Amazon Corretto. Amazon Corretto est une distribution sans coût, multiplateforme et prête à la production de Open Java Development Kit (OpenJDK). Le projet Maven inclut le client d’interface d’exécution en tant que dépendance.

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- Java (par exemple, Amazon Corretto)
- Apache Maven
- AWS CLI version 2
- Docker (version minimale 25.0.0)
- Le plug-in buildx Docker.

Création d’une image à partir d’une image de base alternative

1. Créez un projet Maven. Les paramètres suivants sont obligatoires :

- groupId : l'espace de noms complet du package de votre application.
- artifactId : le nom de votre projet. Cette valeur devient le nom du répertoire de votre projet.

Linux/macOS

```
mvn -B archetype:generate \
   -DarchetypeArtifactId=maven-archetype-quickstart \
   -DgroupId=example \
   -DartifactId=myapp \
   -DinteractiveMode=false
```

PowerShell

```
mvn -B archetype:generate `
   -DarchetypeArtifactId=maven-archetype-quickstart `
   -DgroupId=example `
   -DartifactId=myapp `
   -DinteractiveMode=false
```

2. Ouvrez le répertoire du projet.

```
cd myapp
```

3. Ouvrez le fichier `pom.xml` et remplacez le contenu par ce qui suit. Ce fichier inclut le aws-lambda-java-runtime-interface-client en tant que dépendance. Vous pouvez également installer le client de l'interface d'exécution dans le Dockerfile. Toutefois, l'approche la plus simple consiste à inclure la bibliothèque en tant que dépendance.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>example</groupId>
  <artifactId>hello-lambda</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>hello-lambda</name>
  <url>http://maven.apache.org</url>
  <properties>
```

```
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-runtime-interface-client</artifactId>
      <version>2.3.2</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-dependency-plugin</artifactId>
        <version>3.1.2</version>
        <executions>
          <execution>
            <id>copy-dependencies</id>
            <phase>package</phase>
            <goals>
              <goal>copy-dependencies</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

4. Ouvrez le répertoire *myapp*`/src/main/java/com/example/`*myapp* et trouvez le fichier `App.java`. Il s'agit du code de la fonction Lambda. Remplacez le code par ce qui suit.

   Example gestionnaire de fonctions

```
package example;

public class App {
    public static String sayHello() {
        return "Hello world!";
    }
}
```

5. La commande `mvn -B archetype:generate` de l'étape 1 a également généré un cas de test fictif dans le répertoire `src/test`. Pour les besoins de ce tutoriel, passez outre l'exécution des tests en supprimant l'intégralité de ce répertoire `/test` généré.
6. Retournez au répertoire racine du projet, puis créez un nouveau Dockerfile. L'exemple de Dockerfile suivant utilise une image Amazon Corretto. Amazon Corretto est une distribution sans coût, multiplateforme et prête à la production d'OpenJDK.

   - Définissez la propriété `FROM` sur l'URI de l'image de base.
   - Définissez le `ENTRYPOINT` sur le module que vous souhaitez que le conteneur Docker exécute lorsqu'il démarre. Dans ce cas, le module est le client d'interface d'exécution.
   - Définir l'argument `CMD` pour le gestionnaire de la fonction Lambda.

   Notez que l'exemple de Dockerfile n'inclut pas d'instruction USER. Lorsque vous déployez une image de conteneur sur Lambda, Lambda définit automatiquement un utilisateur Linux par défaut disposant d'autorisations de moindre privilège. Ceci est différent du comportement standard de Docker qui est défini par défaut par l'utilisateur `root` lorsqu'aucune instruction `USER` n'est fournie.

   Example Dockerfile

```
FROM public.ecr.aws/amazoncorretto/amazoncorretto:21 as base

# Configure the build environment
FROM base as build
RUN yum install -y maven
WORKDIR /src

# Cache and copy dependencies
ADD pom.xml .
RUN mvn dependency:go-offline dependency:copy-dependencies

# Compile the function
ADD . .
RUN mvn package

# Copy the function artifact and dependencies onto a clean base
FROM base
WORKDIR /function
```

```
COPY --from=build /src/target/dependency/*.jar ./
COPY --from=build /src/target/*.jar ./

# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "/usr/bin/java", "-cp", "./*",
 "com.amazonaws.services.lambda.runtime.api.client.AWSLambda" ]
# Pass the name of the function handler as an argument to the runtime
CMD [ "example.App::sayHello" ]
```

7. Générez l'image Docker à l'aide de la commande docker build. L'exemple suivant nomme l'image `docker-image` et lui donne la balise `test`. Pour rendre votre image compatible avec Lambda, vous devez utiliser l'option `--provenance=false`.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> La commande spécifie l'option `--platform linux/amd64` pour garantir la compatibilité de votre conteneur avec l'environnement d'exécution Lambda, quelle que soit l'architecture de votre machine de génération. Si vous avez l'intention de créer une fonction Lambda à l'aide de l'architecture du jeu ARM64 d'instructions, veillez à modifier la commande pour utiliser l'`--platform linux/arm64`option à la place.

(Facultatif) Testez l'image localement

Utilisez l'émulateur d'interface d'exécution pour tester l'image localement. Vous pouvez intégrer l'émulateur dans votre image ou utiliser la procédure suivante pour l'installer sur votre machine locale.

Pour installer et exécuter l'émulateur d'interface d'exécution sur votre ordinateur local

1. Depuis le répertoire de votre projet, exécutez la commande suivante pour télécharger l'émulateur d'interface d'exécution (architecture x86-64) GitHub et l'installer sur votre machine locale.

   Linux/macOS

```
mkdir -p ~/.aws-lambda-rie && \
```

```
    curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-
runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
    chmod +x ~/.aws-lambda-rie/aws-lambda-rie
```

Pour installer l'émulateur arm64, remplacez l'URL du GitHub référentiel dans la commande précédente par la suivante :

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

PowerShell

```
$dirPath = "$HOME\.aws-lambda-rie"
if (-not (Test-Path $dirPath)) {
    New-Item -Path $dirPath -ItemType Directory
}

$downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/
releases/latest/download/aws-lambda-rie"
$destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

Pour installer l'émulateur arm64, remplacez `$downloadLink` par ce qui suit :

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. Démarrez votre image Docker à l'aide de la commande docker run. Notez ce qui suit :

   - `docker-image` est le nom de l'image et `test` est la balise.
   - `/usr/bin/java -cp './*' com.amazonaws.services.lambda.runtime.api.client.AWSLambda example.App::sayHello` est le `ENTRYPOINT` suivi du `CMD` depuis votre Dockerfile.

   Linux/macOS

```
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
 9000:8080 \
    --entrypoint /aws-lambda/aws-lambda-rie \
    docker-image:test \
```

```
        /usr/bin/java -cp './*'
 com.amazonaws.services.lambda.runtime.api.client.AWSLambda
 example.App::sayHello
```

PowerShell

```
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
 9000:8080 `
--entrypoint /aws-lambda/aws-lambda-rie `
docker-image:test `
    /usr/bin/java -cp './*'
 com.amazonaws.services.lambda.runtime.api.client.AWSLambda
 example.App::sayHello
```

Cette commande exécute l'image en tant que conteneur et crée un point de terminaison local à `localhost:9000/2015-03-31/functions/function/invocations`.

> **Note**
>
> Si vous avez créé l'image Docker pour l'architecture du jeu ARM64 d'instructions, veillez à utiliser l'`--platform linux/arm64`option au lieu de`--platform linux/amd64`.

3. Publiez un événement au point de terminaison local.

Linux/macOS

Sous Linux et macOS, exécutez la commande `curl` suivante :

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

Cette commande invoque la fonction avec un événement vide et renvoie une réponse. Si vous utilisez votre propre code de fonction plutôt que l'exemple de code de fonction, vous pouvez invoquer la fonction avec une charge utile JSON. Exemple :

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

Dans PowerShell, exécutez la `Invoke-WebRequest` commande suivante :

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Cette commande invoque la fonction avec un événement vide et renvoie une réponse. Si vous utilisez votre propre code de fonction plutôt que l'exemple de code de fonction, vous pouvez invoquer la fonction avec une charge utile JSON. Exemple :

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. Obtenez l'ID du conteneur.

```
docker ps
```

5. Utilisez la commande docker kill pour arrêter le conteneur. Dans cette commande, remplacez `3766c4ab331c` par l'ID du conteneur de l'étape précédente.

```
docker kill 3766c4ab331c
```

Déploiement de l'image

Pour charger l'image sur Amazon RIE et créer la fonction Lambda

1. Exécutez la get-login-passwordcommande pour authentifier la CLI Docker auprès de votre registre Amazon ECR.
   - Définissez la `--region` valeur à l' Région AWS endroit où vous souhaitez créer le référentiel Amazon ECR.
   - `111122223333`Remplacez-le par votre Compte AWS identifiant.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Créez un référentiel dans Amazon ECR à l'aide de la commande create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Le référentiel Amazon ECR doit être Région AWS identique à la fonction Lambda.

En cas de succès, vous obtenez une réponse comme celle-ci :

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Copiez le `repositoryUri` à partir de la sortie de l'étape précédente.
4. Exécutez la commande docker tag pour étiqueter votre image locale dans votre référentiel Amazon ECR en tant que dernière version. Dans cette commande :

   - `docker-image:test` est le nom et la balise de votre image Docker. Il s'agit du nom et de la balise de l'image que vous avez spécifiés dans la commande `docker build`.
   - Remplacez <ECRrepositoryUri> par l'`repositoryUri` que vous avez copié. Assurez-vous d'inclure `:latest` à la fin de l'URI.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Exemple :

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Exécutez la commande docker push pour déployer votre image locale dans le référentiel Amazon ECR. Assurez-vous d'inclure `:latest` à la fin de l'URI du référentiel.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Créez un rôle d'exécution pour la fonction, si vous n'en avez pas déjà un. Vous aurez besoin de l'Amazon Resource Name (ARN) du rôle à l'étape suivante.
7. Créez la fonction Lambda. Pour `ImageUri`, indiquez l'URI du référentiel mentionné précédemment. Assurez-vous d'inclure `:latest` à la fin de l'URI.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> Vous pouvez créer une fonction en utilisant une image d'un autre AWS compte, à condition que l'image se trouve dans la même région que la fonction Lambda. Pour de plus amples informations, veuillez consulter Autorisations entre comptes Amazon ECR.

8. Invoquer la fonction.

```
aws lambda invoke --function-name hello-world response.json
```

Vous devriez obtenir une réponse comme celle-ci :

```
{
  "ExecutedVersion": "$LATEST",
```

```
  "StatusCode": 200
}
```

9. Pour voir la sortie de la fonction, consultez le fichier `response.json`.

Pour mettre à jour le code de fonction, vous devez créer à nouveau l'image, télécharger la nouvelle image dans le référentiel Amazon ECR, puis utiliser la update-function-codecommande pour déployer l'image sur la fonction Lambda.

Lambda résout l'étiquette d'image en hachage d'image spécifique. Cela signifie que si vous pointez la balise d'image qui a été utilisée pour déployer la fonction vers une nouvelle image dans Amazon ECR, Lambda ne met pas automatiquement à jour la fonction pour utiliser la nouvelle image.

Pour déployer la nouvelle image sur la même fonction Lambda, vous devez utiliser la update-function-codecommande, même si la balise d'image dans Amazon ECR reste la même. Dans l'exemple suivant, l'option `--publish` crée une version de la fonction à l'aide de l'image du conteneur mise à jour.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Utilisation de couches pour les fonctions Lambda Java

Utilisez les couches Lambda pour empaqueter le code et les dépendances que vous souhaitez réutiliser dans plusieurs fonctions. Les couches contiennent généralement des dépendances de bibliothèque, une exécution personnalisée, ou des fichiers de configuration. La création d'une couche implique trois étapes générales :

1. Empaquetez le contenu de votre couche. Cela signifie créer une archive de fichiers .zip contenant les dépendances que vous souhaitez utiliser dans vos fonctions.
2. Créez la couche dans Lambda.
3. Ajoutez la couche à vos fonctions.

Rubriques

- Empaqueter le contenu de votre couche
- Créer la couche dans Lambda
- Ajouter la couche à votre fonction

Empaqueter le contenu de votre couche

Pour créer une couche, regroupez vos packages dans une archive (fichier .zip) répondant aux exigences suivantes :

- Assurez-vous que la version Java à laquelle Maven ou Gradle fait référence est identique à la version Java de la fonction que vous souhaitez déployer. Par exemple, pour une fonction Java 25, la `mvn -v` commande doit répertorier Java 25 dans la sortie.
- Vos dépendances doivent être stockées dans le répertoire `java/lib`, à la racine du fichier .zip. Pour de plus amples informations, veuillez consulter Chemins d'accès de couche pour chaque exécution Lambda.
- Les packages de votre couche doivent être compatibles avec Linux. Les fonctions Lambda s'exécutent sur Amazon Linux.

Vous pouvez créer des couches contenant des bibliothèques Java tierces ou vos propres modules et packages Java. La procédure suivante utilise Maven. Vous pouvez également utiliser Gradle pour empaqueter le contenu de votre couche.

Créer une couche à l'aide de dépendances Maven

1. Créez un projet Apache Maven avec un fichier `pom.xml` qui définit vos dépendances.

   L'exemple suivant inclut Jackson Databind pour le traitement JSON. La `<build>` section utilise le maven-dependency-pluginpour créer des fichiers JAR distincts pour chaque dépendance au lieu de les regrouper dans un seul uber-jar. Si vous souhaitez créer un uber-jar, utilisez le. maven-shade-plugin

   Example pom.xml

```
<dependencies>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.17.0</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.13.0</version>
            <configuration>
                <source>21</source>
                <target>21</target>
                <release>21</release>
            </configuration>
        </plugin>

        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-dependency-plugin</artifactId>
            <version>3.6.1</version>
            <executions>
                <execution>
                    <id>copy-dependencies</id>
                    <phase>package</phase>
                    <goals>
                        <goal>copy-dependencies</goal>
                    </goals>
```

```
                    <configuration>
                        <outputDirectory>${project.build.directory}/lib</
outputDirectory>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

2. Générez le projet. Cette commande crée tous les fichiers JAR de dépendance dans le répertoire `target/lib/`.

```
mvn clean package
```

3. Créez la structure de répertoires requise pour votre couche :

```
mkdir -p java/lib
```

4. Copiez les fichiers JAR de dépendance dans le répertoire `java/lib` :

```
cp target/lib/*.jar java/lib/
```

5. Compressez le contenu de la couche :

   Linux/macOS

   ```
   zip -r layer.zip java/
   ```

   PowerShell

   ```
   Compress-Archive -Path .\java -DestinationPath .\layer.zip
   ```

   La structure de répertoire de votre fichier .zip devrait se présenter comme suit :

```
java/
### lib/
    ### jackson-databind-2.17.0.jar
    ### jackson-core-2.17.0.jar
    ### jackson-annotations-2.17.0.jar
```

> **Note**
>
> Assurez-vous que votre fichier .zip inclut le répertoire `java` au niveau racine avec `lib` à l'intérieur. Cette structure garantit que Lambda peut localiser et importer vos bibliothèques. Chaque dépendance est conservée dans un fichier JAR distinct plutôt que regroupée dans un fichier uber-jar.

Créer la couche dans Lambda

Vous pouvez publier votre couche à l'aide de la console Lambda AWS CLI ou de la console Lambda.

AWS CLI

Exécutez la publish-layer-version AWS CLI commande pour créer la couche Lambda :

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip
 --compatible-runtimes java25
```

Le paramètre compatible runtimes est facultatif. Lorsqu'il est spécifié, Lambda utilise ce paramètre pour filtrer les couches dans la console Lambda.

Console

Pour créer une couche (console)

1. Ouvrez la page Couches de la console Lambda.
2. Sélectionnez Créer un calque.
3. Choisissez Charger un fichier .zip, puis chargez l'archive .zip que vous avez créée précédemment.
4. (Facultatif) Pour Environnements d'exécution compatibles, choisissez l'environnement d'exécution Java qui correspond à la version de Java que vous avez utilisée pour créer votre couche.
5. Choisissez Créer.

Ajouter la couche à votre fonction

AWS CLI

Pour associer la couche à votre fonction, exécutez la update-function-configuration AWS CLI commande. Utilisez l'ARN de la couche pour le paramètre `--layers`. L'ARN doit spécifier la version (par exemple, `arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`). Pour de plus amples informations, veuillez consulter Couches et versions de couches.

```
aws lambda update-function-configuration --function-name my-function --cli-binary-format raw-in-base64-out --layers "arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1"
```

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

Console

Ajouter une couche à une fonction

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez la fonction.
3. Faites défiler jusqu'à la section Couches, puis choisissez Ajouter une couche.
4. Sous Choisir une couche, sélectionnez Couches personnalisées, puis choisissez votre couche.

   > **Note**
   >
   > Si vous n'avez pas ajouté d'environnement d'exécution compatible lors de la création de la couche, celle-ci ne sera pas répertoriée ici. Vous pouvez spécifier l'ARN de la couche à la place.

5. Choisissez Ajouter.

Personnalisation de la sérialisation pour les fonctions Lambda Java

Les environnements d'exécution gérés par Lambda Java prennent en charge la sérialisation personnalisée pour les événements JSON. La sérialisation personnalisée peut simplifier votre code et potentiellement améliorer les performances.

Rubriques

- Cas d'utilisation de la sérialisation personnalisée
- Implémentation de sérialisation personnalisée
- Test de la sérialisation personnalisée

Cas d'utilisation de la sérialisation personnalisée

Lorsque votre fonction Lambda est invoquée, les données d'événement d'entrée doivent être désérialisées dans un objet Java, et la sortie de votre fonction doit être sérialisée à nouveau dans un format pouvant être renvoyé en tant que réponse de la fonction. Les environnements d'exécution gérés par Lambda Java fournissent des fonctionnalités de sérialisation et de désérialisation par défaut qui fonctionnent bien pour gérer les données utiles d'événements provenant de divers services AWS, tels qu'Amazon API Gateway et Amazon Simple Queue Service (Amazon SQS). Pour utiliser ces événements d'intégration de services dans votre fonction, ajoutez la dépendance aws-java-lambda-events à votre projet. Cette bibliothèque AWS contient des objets Java représentant ces événements d'intégration de services.

Vous pouvez également utiliser vos propres objets pour représenter l'événement JSON que vous transmettez à votre fonction Lambda. L'environnement d'exécution géré tente de sérialiser le JSON vers une nouvelle instance de votre objet avec son comportement par défaut. Si le sérialiseur par défaut n'a pas le comportement souhaité pour votre cas d'utilisation, utilisez la sérialisation personnalisée.

Par exemple, si votre gestionnaire de fonctions attend une classe `Vehicle` en entrée, avec la structure suivante :

```
public class Vehicle {
    private String vehicleType;
    private long vehicleId;
}
```

Cependant, les données utiles de l'événement JSON ressemblent à ceci :

```
{
    "vehicle-type": "car",
    "vehicleID": 123
}
```

Dans ce scénario, la sérialisation par défaut dans l'environnement d'exécution géré s'attend à ce que les noms de propriété JSON correspondent au camel case des noms de propriété de classe Java (`vehicleType`, `vehicleId`). Dans la mesure où les noms des propriétés dans l'événement JSON ne sont pas écrits en camel case (`vehicle-type`,`vehicleID`), vous devez utiliser une sérialisation personnalisée.

Implémentation de sérialisation personnalisée

Utilisez une interface de fournisseur de services pour charger le sérialiseur de votre choix au lieu de la logique de sérialisation par défaut de l'environnement d'exécution géré. Vous pouvez sérialiser vos données utiles d'événements JSON directement dans des objets Java, à l'aide de l'interface standard `RequestHandler`.

Pour utiliser la sérialisation personnalisée dans votre fonction Lambda Java

1. Ajoutez la bibliothèque aws-lambda-java-core en tant que dépendance. Cette bibliothèque inclut l'interface CustomPojoSerializer, ainsi que d'autres définitions d'interface permettant de travailler avec Java dans Lambda.
2. Créez un fichier nommé `com.amazonaws.services.lambda.runtime.CustomPojoSerializer` dans votre répertoire `src/main/resources/META-INF/services/` de votre projet.
3. Dans ce fichier, spécifiez le nom complet de votre implémentation de sérialiseur personnalisé, qui doit implémenter l'interface `CustomPojoSerializer`. Exemple :

   ```
   com.mycompany.vehicles.CustomLambdaSerialzer
   ```

4. Implémentez l'interface `CustomPojoSerializer` pour fournir votre logique de sérialisation personnalisée.
5. Utilisez l'interface standard `RequestHandler` de votre fonction Lambda. L'environnement d'exécution géré utilisera votre sérialiseur personnalisé.

Pour d'avantage d'exemples d'implémenation de la sérialisation personnalisée à l'aide de bibliothèques populaires telles que fastJson, Gson, Moshi et jackson-jr, consultez l'exemple custom-serialization dans le référentiel GitHub AWS.

Test de la sérialisation personnalisée

Testez votre fonction pour vous assurer que votre logique de sérialisation et de désérialisation fonctionne comme prévu. Vous pouvez utiliser l'interface de ligne de commande AWS Serverless Application Model (AWS SAM CLI) pour émuler l'invocation de vos données utiles Lambda. Cela peut vous aider à tester et à itérer rapidement votre fonction lorsque vous introduisez un sérialiseur personnalisé.

1. Créez un fichier avec les données utiles de l'événement JSON avec lesquelles vous souhaitez invoquer votre fonction, puis appelez le AWS SAM CLI.
2. Exécutez la commande sam local invoke pour invoquer votre fonction localement. Exemple :

```
sam local invoke -e src/test/resources/event.json
```

Pour plus d'informations, consultez Locally invoke Lambda functions with AWS SAM.

Personnalisation du comportement de démarrage de l'environnement d'exécution Java pour les fonctions Lambda

Cette page décrit les paramètres spécifiques aux fonctions Java dans AWS Lambda. Vous pouvez utiliser ces paramètres pour personnaliser le comportement de démarrage de l'exécution Java. Cela permet de réduire le temps de latence et d'améliorer les performances globales de la fonction, sans avoir à modifier le code.

Sections

- Présentation de la variable d'environnement JAVA_TOOL_OPTIONS
- Patch Log4j pour Log4Shell
- Ahead-of-Time (AOT) et caches CDS

Présentation de la variable d'environnement **`JAVA_TOOL_OPTIONS`**

Sur Java, Lambda prend en charge la variable d'environnement `JAVA_TOOL_OPTIONS` pour définir des variables de ligne de commande supplémentaires dans Lambda. Vous pouvez utiliser cette variable d'environnement de différentes manières, par exemple pour personnaliser les paramètres de compilation par niveaux. L'exemple suivant montre comment utiliser la variable d'environnement `JAVA_TOOL_OPTIONS` pour ce cas d'utilisation.

Exemple : personnalisation des paramètres de compilation par niveaux

La compilation par niveaux est une fonctionnalité de la JVM (Java Virtual Machine). Vous pouvez utiliser des paramètres de compilation hiérarchisés spécifiques pour tirer le meilleur parti des compilateurs de la JVM just-in-time (JIT). Généralement, le compilateur C1 est optimisé pour un démarrage rapide. Le compilateur C2 est optimisé pour obtenir les meilleures performances globales, mais il utilise également plus de mémoire et prend plus de temps pour y parvenir. Il existe cinq niveaux différents de compilation par niveaux. Au niveau 0, la JVM interprète le bytecode Java. Au niveau 4, la JVM utilise le compilateur C2 pour analyser les données de profilage collectées lors du démarrage de l'application. Au fil du temps, elle surveille l'utilisation du code pour identifier les meilleures optimisations.

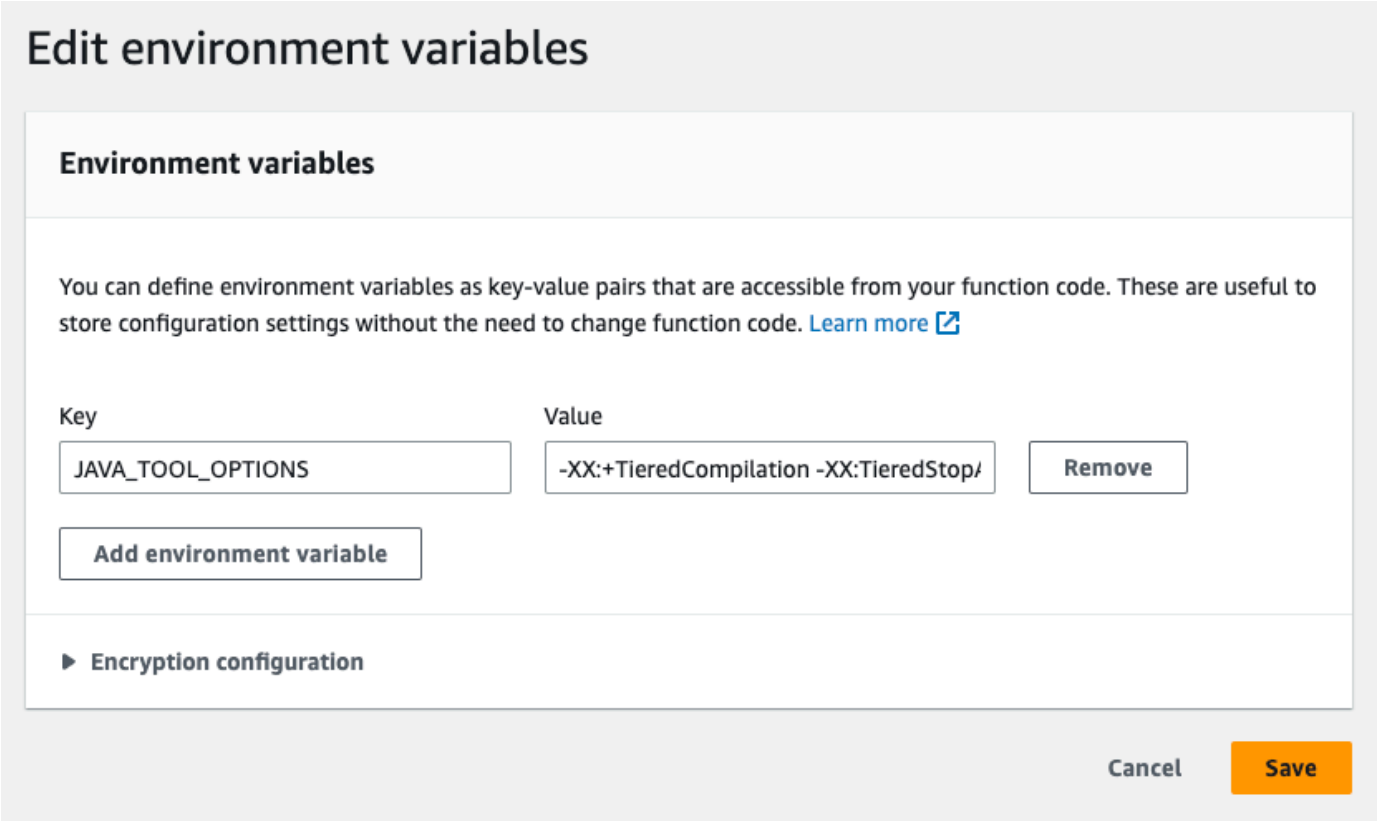
La personnalisation du niveau de compilation à plusieurs niveaux peut vous aider à optimiser les performances de vos fonctions Java. Pour les petites fonctions qui s'exécutent rapidement, le réglage de la compilation hiérarchisée au niveau 1 peut contribuer à améliorer les performances de

démarrage à froid en faisant en sorte que la JVM utilise le compilateur C1. Ce paramètre produit rapidement du code natif optimisé, mais il ne génère aucune donnée de profilage et n'utilise jamais le compilateur C2. Pour les fonctions plus volumineuses nécessitant des calculs intensifs, le fait de définir la compilation hiérarchisée au niveau 4 maximise les performances globales au détriment d'une consommation de mémoire supplémentaire et d'un travail d'optimisation supplémentaire lors des premiers appels après le provisionnement de chaque environnement d'exécution Lambda.

Pour les environnements d'exécution Java 11 et inférieurs, Lambda utilise les paramètres de compilation hiérarchisée par défaut de la JVM. Pour Java 17 et Java 21, Lambda configure la JVM pour arrêter la compilation hiérarchisée au niveau 1 par défaut. À partir de Java 25, Lambda arrête toujours la compilation hiérarchisée au niveau 1 par défaut, sauf en cas d'utilisation SnapStart ou de provisionnement de la concurrence, auquel cas les paramètres JVM par défaut sont utilisés. Cela améliore les performances SnapStart et la simultanéité provisionnée sans entraîner de pénalité de démarrage à froid, car la compilation à plusieurs niveaux est effectuée en dehors du chemin d'appel dans ces cas. Pour optimiser cet avantage, vous pouvez utiliser l'amorçage, c'est-à-dire l'exécution de chemins de code lors de l'initialisation de la fonction pour déclencher le JIT avant de prendre le SnapStart snapshot ou lorsque les environnements d'exécution de la concurrence provisionnée sont préprovisionnés. Pour plus d'informations, consultez le billet de blog Optimisation des performances de démarrage à froid de AWS Lambda à l'aide de stratégies d'amorçage avancées avec. SnapStart

Pour personnaliser les paramètres de compilation par niveaux (console)

1. Ouvrez la page Fonctions de la console Lambda.
2. Choisissez une fonction Java pour laquelle vous souhaitez personnaliser la compilation par niveau.
3. Choisissez l'onglet Configuration, puis Variables d'environnement dans le menu de gauche.
4. Choisissez Modifier.
5. Choisissez Ajouter une variable d’environnement.
6. Pour Nom de la clé, saisissez `JAVA_TOOL_OPTIONS`. Pour Valeur, saisissez `-XX:+TieredCompilation -XX:TieredStopAtLevel=1`.



7. Choisissez Enregistrer.

> **Note**
>
> Vous pouvez également utiliser Lambda SnapStart pour atténuer les problèmes de démarrage à froid. SnapStartutilise des instantanés mis en cache de votre environnement d'exécution pour améliorer de manière significative les performances de démarrage. Pour plus d'informations sur les SnapStart fonctionnalités, les limitations et les régions prises en charge, consultezAméliorer les performances de démarrage avec Lambda SnapStart.

Exemple : personnalisation du comportement de GC à l'aide de JAVA_TOOL_OPTIONS

Les exécutions Java 11 utilisent le récupérateur de mémoire (GC) Serial pour la récupération de mémoire. Par défaut, les exécutions Java 17 utilisent également le GC Serial. Toutefois, avec Java 17, vous pouvez également utiliser la variable d'environnement `JAVA_TOOL_OPTIONS` pour modifier le GC par défaut. Vous pouvez choisir entre le GC Parallel et le GC Shenandoah.

Par exemple, si votre charge de travail utilise plus de mémoire et de multiples CPUs, pensez à utiliser le Parallel GC pour de meilleures performances. Vous pouvez le faire en ajoutant ce qui suit à la valeur de votre variable d'environnement `JAVA_TOOL_OPTIONS` :

```
-XX:+UseParallelGC
```

Si votre charge de travail comporte de nombreux objets de courte durée, vous pouvez bénéficier d'une réduction de la consommation de mémoire en activant le mode générationnel du ramasse-miettes Shenandoah introduit dans Java 25. Pour ce faire, ajoutez ce qui suit à la valeur de votre variable d'`JAVA_TOOL_OPTIONS`environnement :

```
-XX:+UseShenandoahGC -XX:ShenandoahGCMode=generational
```

Patch Log4j pour Log4Shell

Les environnements d'exécution Lambda pour Java 8, 11, 17 et 21 incluent un correctif pour atténuer la vulnérabilité Log4Shell (CVE-2021-44228) dans Log4j, un framework de journalisation Java populaire. Ce correctif entraîne une surcharge des performances de démarrage à froid. Si vous utilisez une version corrigée de Log4j (version 2.17.0 ou ultérieure), vous pouvez désactiver ce correctif pour améliorer les performances de démarrage à froid. Pour désactiver le correctif, définissez la variable d'`AWS_LAMBDA_DISABLE_CVE_2021_44228_PROTECTION`environnement sur`true`.

À partir de Java 25, les environnements d'exécution Lambda n'incluent plus le correctif Log4Shell. Vous devez vérifier que vous utilisez Log4j version 2.17.0 ou ultérieure.

Ahead-of-Time (AOT) et caches CDS

À partir de Java 25, le moteur d'exécution Lambda inclut un cache Ahead-of-Time (AOT) pour le client d'interface d'exécution Java (RIC), un composant d'exécution qui interroge activement les événements provenant de l'API Lambda Runtime. Cela améliore les performances de démarrage à froid.

Les caches AOT sont spécifiques à une version de machine virtuelle Java. Lorsque Lambda met à jour le runtime géré, il met également à jour le cache AOT du RIC. Toutefois, si vous déployez vos propres caches AOT, ceux-ci peuvent être invalidés ou entraîner un comportement inattendu après une mise à jour d'exécution. Nous recommandons donc vivement de ne pas utiliser les caches AOT

lorsque vous utilisez des environnements d'exécution gérés. Pour utiliser les caches AOT, vous devez déployer vos fonctions à l'aide d'images de conteneur.

Les caches AOT ne peuvent pas être utilisés avec les caches de partage de données de classe (CDS). Si vous déployez des caches CDS dans votre fonction Lambda, Lambda désactive le cache AOT.

Utilisation de l’objet de contexte Lambda pour récupérer les informations de la fonction Java

Lorsque Lambda exécute votre fonction, il transmet un objet contexte au gestionnaire. Cet objet fournit des méthodes et des propriétés fournissant des informations sur l’invocation, la fonction et l’environnement d’exécution.

Méthodes de contexte

- `getRemainingTimeInMillis()` – Renvoie le nombre de millisecondes restant avant l'expiration de l'exécution.
- `getFunctionName()` – Renvoie le nom de la fonction Lambda.
- `getFunctionVersion()` – Renvoie la version de la fonction.
- `getInvokedFunctionArn()` – Renvoie l'Amazon Resource Name (ARN) utilisé pour invoquer la fonction. Indique si l'appelant a spécifié un numéro de version ou un alias.
- `getMemoryLimitInMB()` – Renvoie la quantité de mémoire allouée à la fonction.
- `getAwsRequestId()` – Renvoie l'identifiant de la demande d'invocation.
- `getLogGroupName()` – Renvoie le groupe de journaux pour la fonction.
- `getLogStreamName()` – Renvoie le flux de journal de l'instance de fonction.
- `getIdentity()` – (applications mobiles) Renvoie des informations sur l'identité Amazon Cognito qui a autorisé la demande.
- `getClientContext()` – (applications mobiles) Renvoie le contexte client fourni à Lambda par l'application client.
- `getLogger()` – Renvoie l'objet enregistreur pour la fonction.

L'exemple suivant montre une fonction qui utilise l'objet contexte pour accéder à l'enregistreur Lambda.

Example handler.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
```

```
import com.amazonaws.services.lambda.runtime.RequestHandler;

import java.util.Map;

// Handler value: example.Handler
public class Handler implements RequestHandler<Map<String,String>, Void>{

  @Override
  public Void handleRequest(Map<String,String> event, Context context)
  {
    LambdaLogger logger = context.getLogger();
    logger.log("EVENT TYPE: " + event.getClass());
    return null;
  }
}
```

La fonction enregistre le type de classe de l'événement entrant avant de le renvoyer `null`.

Example sortie de journal

```
EVENT TYPE: class java.util.LinkedHashMap
```

L'interface de l'objet contexte est disponible dans la bibliothèque aws-lambda-java-core. Vous pouvez implémenter cette interface pour créer une classe de contexte à des fins de test. L'exemple suivant montre une classe de contexte qui renvoie des valeurs factices pour la plupart des propriétés et un enregistreur de test opérationnel.

Example src/test/java/example/TestContext.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.CognitoIdentity;
import com.amazonaws.services.lambda.runtime.ClientContext;
import com.amazonaws.services.lambda.runtime.LambdaLogger;

public class TestContext implements Context{

  public TestContext() {}
  public String getAwsRequestId(){
    return new String("495b12a8-xmpl-4eca-8168-160484189f99");
  }
```

```
  public String getLogGroupName(){
    return new String("/aws/lambda/my-function");
  }
  public String getLogStreamName(){
    return new String("2020/02/26/[$LATEST]704f8dxmpla04097b9134246b8438f1a");
  }
  public String getFunctionName(){
    return new String("my-function");
  }
  public String getFunctionVersion(){
    return new String("$LATEST");
  }
  public String getInvokedFunctionArn(){
    return new String("arn:aws:lambda:us-east-2:123456789012:function:my-function");
  }
  public CognitoIdentity getIdentity(){
    return null;
  }
  public ClientContext getClientContext(){
    return null;
  }
  public int getRemainingTimeInMillis(){
    return 300000;
  }
  public int getMemoryLimitInMB(){
    return 512;
  }
  public LambdaLogger getLogger(){
    return new TestLogger();
  }

}
```

Pour de plus amples informations sur la journalisation, veuillez consulter Journalisation et surveillance des fonctions Lambda Java.

Contexte dans des exemples d'applications

Le référentiel GitHub de ce guide comprend des exemples d'applications qui démontrent l'utilisation de l'objet contextuel. Chaque exemple d'application inclut des scripts facilitant le déploiement et le nettoyage, un modèle AWS Serverless Application Model (AWS SAM) et des ressources de support.

Exemples d’applications Lambda en Java

- example-java : fonction Java qui montre comment utiliser Lambda pour traiter les commandes. Cette fonction montre comment définir et désérialiser un objet d’événement d’entrée personnalisé, utiliser le SDK AWS et enregistrer les sorties.
- java-basic – Ensemble de fonctions Java minimales avec des tests unitaires et une configuration de journalisation variable.
- java events – Ensemble de fonctions Java contenant du code squelette permettant de gérer les événements de divers services tels qu’Amazon API Gateway, Amazon SQS et Amazon Kinesis. Ces fonctions utilisent la dernière version (3.0.0 et ultérieure) de la bibliothèque aws-lambda-java-events. Ces exemples ne nécessitent pas le kit SDK AWS comme dépendance.
- s3-java – Fonction Java qui traite les événements de notification d’Amazon S3 et utilise la bibliothèque de classes Java (JCL) pour créer des miniatures à partir de fichiers d’image chargés.
- layer-java : fonction Java qui illustre comment utiliser une couche Lambda pour empaqueter les dépendances séparément du code de votre fonction principale.

Journalisation et surveillance des fonctions Lambda Java

AWS Lambda surveille automatiquement les fonctions Lambda et envoie des entrées de journal à Amazon. CloudWatch Votre fonction Lambda est fournie avec un groupe de CloudWatch journaux Logs et un flux de journaux pour chaque instance de votre fonction. L'environnement d'exécution Lambda envoie des détails sur chaque invocation et d'autres sorties provenant du code de votre fonction au flux de journaux. Pour plus d'informations sur CloudWatch les journaux, consultezEnvoi automatique des journaux de la fonction Lambda à CloudWatch Logs.

Pour produire des journaux à partir de votre code de fonction, vous pouvez utiliser des méthodes sur java.lang.System, ou tout module de journalisation qui écrit sur stdout ou stderr.

Sections

- Création d'une fonction qui renvoie des journaux
- Utilisation des contrôles de journalisation avancés de Lambda avec Java
- Implémentation de la journalisation avancée avec Log4j2 et J SLF4
- Utilisation d'autres outils et bibliothèques de journalisation
- Utilisation de Powertools pour AWS Lambda (Java) et AWS SAM pour la journalisation structurée
- Affichage des journaux dans la console Lambda
- Afficher les journaux dans la CloudWatch console
- Afficher les journaux à l'aide de AWS Command Line Interface (AWS CLI)
- Suppression de journaux
- Exemple de code de journalisation

Création d'une fonction qui renvoie des journaux

Pour générer les journaux à partir de votre code de fonction, vous pouvez utiliser des méthodes sur java.lang.System ou n'importe quelle bibliothèque de journalisation qui écrit dans `stdout` ou `stderr`. La aws-lambda-java-corebibliothèque fournit une classe d'enregistreur nommée à `LambdaLogger` laquelle vous pouvez accéder depuis l'objet de contexte. La classe d'enregistreur prend en charge les journaux multilignes.

L'exemple suivant utilise l'enregistreur `LambdaLogger` fourni par l'objet de contexte.

Example handler.java

```
// Handler value: example.Handler
public class Handler implements RequestHandler<Object, String>{
  Gson gson = new GsonBuilder().setPrettyPrinting().create();
  @Override
  public String handleRequest(Object event, Context context)
  {
    LambdaLogger logger = context.getLogger();
    String response = new String("SUCCESS");
    // log execution details
    logger.log("ENVIRONMENT VARIABLES: " + gson.toJson(System.getenv()));
    logger.log("CONTEXT: " + gson.toJson(context));
    // process event
    logger.log("EVENT: " + gson.toJson(event));
    return response;
  }
}
```

Example format des journaux

```
START RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0 Version: $LATEST
ENVIRONMENT VARIABLES:
{
    "_HANDLER": "example.Handler",
    "AWS_EXECUTION_ENV": "AWS_Lambda_java8",
    "AWS_LAMBDA_FUNCTION_MEMORY_SIZE": "512",
    ...
}
CONTEXT:
{
    "memoryLimit": 512,
    "awsRequestId": "6bc28136-xmpl-4365-b021-0ce6b2e64ab0",
    "functionName": "java-console",
    ...
}
EVENT:
{
  "records": [
    {
      "messageId": "19dd0b57-xmpl-4ac1-bd88-01bbb068cb78",
      "receiptHandle": "MessageReceiptHandle",
      "body": "Hello from SQS!",
```

```
        ...
    }
  ]
}
END RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0
REPORT RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0 Duration: 198.50 ms Billed
 Duration: 724 ms Memory Size: 512 MB Max Memory Used: 90 MB Init Duration: 524.75 ms
```

L'environnement d'exécution Java enregistre les lignes `START`, `END` et `REPORT` pour chaque invocation. La ligne de rapport fournit les détails suivants.

Champs de données de la ligne REPORT

- RequestId— L'identifiant de demande unique pour l'invocation.
- Duration – Temps que la méthode de gestion du gestionnaire de votre fonction a consacré au traitement de l'événement.
- Billed Duration : temps facturé pour l'invocation.
- Memory Size – Quantité de mémoire allouée à la fonction.
- Max Memory Used – Quantité de mémoire utilisée par la fonction. Lorsque les appels partagent un environnement d'exécution, Lambda indique la mémoire maximale utilisée pour toutes les invocations. Ce comportement peut entraîner une valeur signalée plus élevée que prévu.
- Init Duration : pour la première requête servie, temps qu'il a pris à l'exécution charger la fonction et exécuter le code en dehors de la méthode du gestionnaire.
- XRAY TraceId — Pour les demandes suivies, l'ID de AWS X-Ray trace.
- SegmentId— Pour les demandes tracées, l'identifiant du segment X-Ray.
- Sampled : pour les demandes suivies, résultat de l'échantillonnage.

Utilisation des contrôles de journalisation avancés de Lambda avec Java

Pour mieux contrôler la manière dont les journaux de vos fonctions sont capturés, traités et consommés, vous pouvez configurer les options de journalisation suivantes pour les environnements d'exécution Java pris en charge :

- Format de journal : choisissez entre le format texte brut et le format JSON structuré pour les journaux de votre fonction
- Niveau du journal : pour les journaux au format JSON, choisissez le niveau de détail des journaux auxquels Lambda envoie CloudWatch, par exemple ERROR, DEBUG ou INFO

- Groupe de journaux : choisissez le groupe de CloudWatch journaux auquel votre fonction envoie les journaux

Pour plus d'informations sur ces options de journalisation et pour savoir comment configurer votre fonction pour les utiliser, consultez the section called "Configuration de commandes de journalisation avancées pour votre fonction Lambda".

Pour utiliser le format de journal et les options de niveau de journal avec vos fonctions Java Lambda, consultez les instructions des sections suivantes.

Utilisation du format de journal JSON structuré avec Java

Si vous sélectionnez JSON pour le format de journal de votre fonction, Lambda envoie les journaux en utilisant la classe `LambdaLogger` sous forme de JSON structuré. Chaque objet de journal JSON contient au moins quatre paires clé-valeur avec les clés suivantes :

- `"timestamp"` - heure à laquelle le message de journal a été généré
- `"level"` - niveau de journalisation attribué au message
- `"message"` - contenu du message de journal
- `"AWSrequestId"` - identifiant unique de la demande pour l'invocation de la fonction

Selon la méthode de journalisation que vous utilisez, les sorties de journal de votre fonction capturées au format JSON peuvent également contenir des paires clé-valeur supplémentaires.

Pour attribuer un niveau aux journaux que vous créez à l'aide de l'enregistreur `LambdaLogger`, vous devez fournir un argument `LogLevel` dans votre commande de journalisation, comme illustré dans l'exemple suivant.

Example Code de journalisation Java

```
LambdaLogger logger = context.getLogger();
logger.log("This is a debug log", LogLevel.DEBUG);
```

Cette sortie de journal produite par cet exemple de code serait capturée dans CloudWatch Logs comme suit :

Example Enregistrement de journaux JSON

```
{
```

```
    "timestamp":"2023-11-01T00:21:51.358Z",
    "level":"DEBUG",
    "message":"This is a debug log",
    "AWSrequestId":"93f25699-2cbf-4976-8f94-336a0aa98c6f"
}
```

Si vous n'attribuez aucun niveau à la sortie de votre journal, Lambda lui attribuera automatiquement le niveau INFO.

Si votre code utilise déjà une autre bibliothèque de journaux pour produire des journaux structurés JSON, vous n'avez pas besoin d'apporter de modifications. Lambda ne double code aucun journal déjà codé au format JSON. Même si vous configurez votre fonction pour utiliser le format de journal JSON, vos sorties de journalisation apparaissent CloudWatch dans la structure JSON que vous définissez.

Utilisation du filtrage au niveau du journal avec Java

AWS Lambda Pour filtrer les journaux de votre application en fonction de leur niveau de journalisation, votre fonction doit utiliser des journaux au format JSON. Vous pouvez effectuer cette opération de deux façons :

- Créez des sorties de journal à l'aide de la norme `LambdaLogger` et configurez votre fonction pour utiliser le format de journal JSON. Lambda filtre ensuite les sorties de votre journal à l'aide de la paire clé-valeur « niveau » de l'objet JSON décrit dans the section called "Utilisation du format de journal JSON structuré avec Java". Pour savoir comment configurer le format de journal de votre fonction, consultez the section called "Configuration de commandes de journalisation avancées pour votre fonction Lambda".
- Utilisez une autre bibliothèque ou méthode de journalisation pour créer des journaux structurés JSON dans votre code qui incluent une paire clé-valeur « niveau » définissant le niveau de sortie du journal. Vous pouvez utiliser n'importe quelle bibliothèque de journalisation qui écrit des journaux JSON dans `stdout` ou `stderr`. Par exemple, vous pouvez utiliser Powertools for AWS Lambda ou le package Log4j2 pour générer des sorties de journal structurées JSON à partir de votre code. Pour en savoir plus, consultez the section called "Utilisation de Powertools pour AWS Lambda (Java) et AWS SAM pour la journalisation structurée" et the section called "Implémentation de la journalisation avancée avec Log4j2 et J SLF4".

Lorsque vous configurez votre fonction pour utiliser le filtrage au niveau des journaux, vous devez sélectionner l'une des options suivantes pour le niveau de journaux que Lambda doit envoyer à Logs : CloudWatch

Niveau du journal	Utilisation standard
TRACE (le plus détaillé)	Les informations les plus précises utilisées pour tracer le chemin d'exécution de votre code
DEBUG	Informations détaillées pour le débogage du système
INFO	Messages qui enregistrent le fonctionnement normal de votre fonction
WARN	Messages relatifs à des erreurs potentielles susceptibles d'entraîner un comportement inattendu si elles ne sont pas traitées
ERROR	Messages concernant les problèmes qui empêchent le code de fonctionner comme prévu
FATAL (moindre détail)	Messages relatifs à des erreurs graves entraînant l'arrêt du fonctionnement de l'applica tion

Pour que Lambda puisse filtrer les journaux de votre fonction, vous devez également inclure une paire `"timestamp"` clé-valeur dans la sortie de votre journal JSON. L'heure doit être spécifiée dans un format d'horodatage RFC 3339 valide. Si vous ne fournissez pas d'horodatage valide, Lambda attribuera au journal le niveau INFO et ajoutera un horodatage pour vous.

Lambda envoie les journaux du niveau sélectionné et inférieur à. CloudWatch Par exemple, si vous configurez un niveau de journalisation WARN, Lambda envoie des journaux correspondant aux niveaux WARN, ERROR et FATAL.

Implémentation de la journalisation avancée avec Log4j2 et J SLF4

> **Note**
>
> AWS Lambda n'inclut pas Log4j2 dans ses environnements d'exécution gérés ni dans ses images de conteneur de base. Ces problèmes ne sont donc pas affectés par les problèmes décrits dans CVE-2021-44228, CVE-2021-45046 et CVE-2021-45105.
> Dans les cas où une fonction client comprend une version de Log4j2 affectée, nous avons appliqué un changement aux exécutions gérées par Lambda Java et aux images de conteneur de base qui permet d'atténuer les problèmes liés à CVE-2021-44228, CVE-2021-45046 et CVE-2021-45105. À la suite de cette modification, les clients utilisant Log4J2 peuvent voir une entrée de journal supplémentaire, similaire à « `Transforming org/apache/logging/log4j/core/lookup/JndiLookup (java.net.URLClassLoader@...)` ». Toute chaîne de journal qui fait référence au mappeur jndi dans la sortie Log4J2 sera remplacée par « `Patched JndiLookup::lookup()` ».
> Indépendamment de ce changement, nous encourageons vivement tous les clients dont les fonctions incluent Log4j2 à mettre à jour vers la dernière version. Plus précisément, les clients utilisant la bibliothèque aws-lambda-java-log 4j2 dans leurs fonctions doivent passer à la version 1.5.0 (ou ultérieure) et redéployer leurs fonctions. Cette version met à jour les dépendances sous-jacentes de l'utilitaire Log4j2 vers la version 2.17.0 (ou ultérieure). Le binaire aws-lambda-java-log 4j2 mis à jour est disponible dans le dépôt Maven et son code source est disponible sur Github.
> Enfin, notez que les bibliothèques liées à aws-lambda-java-log4j (v1.0.0 ou 1.0.1) ne doivent en aucun cas être utilisées. Ces bibliothèques sont liées à la version 1.x de log4j qui a pris fin en 2015. Les bibliothèques ne sont pas prises en charge, ne sont pas maintenues, ne sont pas corrigées et présentent des failles de sécurité connues.

Pour personnaliser la sortie du journal, prendre en charge la journalisation pendant les tests unitaires et enregistrer les appels du AWS SDK, utilisez Apache Log4j2 avec SLF4 J. Log4j est une bibliothèque de journalisation pour les programmes Java qui vous permet de configurer les niveaux de journalisation et d'utiliser des bibliothèques d'annexes. SLF4J est une bibliothèque de façade qui vous permet de changer la bibliothèque que vous utilisez sans changer le code de votre fonction.

Pour ajouter l'ID de demande aux journaux de votre fonction, utilisez l'annexe de la bibliothèque aws-lambda-java-log4j2.

Example src/main/resources/log4j2.xml — Configuration de l'annexe

```
<Configuration>
  <Appenders>
    <Lambda name="Lambda" format="${env:AWS_LAMBDA_LOG_FORMAT:-TEXT}">
       <LambdaTextFormat>
         <PatternLayout>
             <pattern>%d{yyyy-MM-dd HH:mm:ss} %X{AWSRequestId} %-5p %c{1} - %m%n </
pattern>
         </PatternLayout>
       </LambdaTextFormat>
       <LambdaJSONFormat>
         <JsonTemplateLayout eventTemplateUri="classpath:LambdaLayout.json" />
       </LambdaJSONFormat>
     </Lambda>
   </Appenders>
   <Loggers>
     <Root level="${env:AWS_LAMBDA_LOG_LEVEL:-INFO}">
       <AppenderRef ref="Lambda"/>
     </Root>
     <Logger name="software.amazon.awssdk" level="WARN" />
     <Logger name="software.amazon.awssdk.request" level="DEBUG" />
   </Loggers>
 </Configuration>
```

Vous pouvez décider comment vos journaux Log4j2 sont configurés pour les sorties en texte brut ou JSON en spécifiant une mise en page sous les balises `<LambdaTextFormat>` et `<LambdaJSONFormat>`.

Dans cet exemple, en mode texte, chaque ligne est précédée de la date, de l'heure, de l'ID de demande, du niveau de journal et du nom de classe. En mode JSON, `<JsonTemplateLayout>` est utilisé avec une configuration fournie avec la bibliothèque `aws-lambda-java-log4j2`.

SLF4J est une bibliothèque de façade permettant de se connecter au code Java. Dans votre code de fonction, vous utilisez la SLF4 J logger factory pour récupérer un enregistreur avec des méthodes pour les niveaux de journalisation telles que `info()` et. `warn()` Dans votre configuration de compilation, vous incluez la bibliothèque de journalisation et l'adaptateur SLF4 J dans le chemin de classe. En modifiant les bibliothèques dans la configuration de construction, vous pouvez modifier le type d'enregistreur sans modifier votre code de fonction. SLF4J est requis pour capturer les journaux à partir du SDK for Java.

Dans l'exemple de code suivant, la classe de gestionnaire utilise SLF4 J pour récupérer un enregistreur.

Example src/main/java/example/HandlerS3.java — Journalisation avec SLF4 J

```
package example;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;

import static org.apache.logging.log4j.CloseableThreadContext.put;


public class HandlerS3 implements RequestHandler<S3Event, String>{
    private static final Logger logger = LoggerFactory.getLogger(HandlerS3.class);

    @Override
    public String handleRequest(S3Event event, Context context) {
        for(var record : event.getRecords()) {
            try (var loggingCtx = put("awsRegion", record.getAwsRegion())) {
                loggingCtx.put("eventName", record.getEventName());
                loggingCtx.put("bucket", record.getS3().getBucket().getName());
                loggingCtx.put("key", record.getS3().getObject().getKey());

                logger.info("Handling s3 event");
            }
        }

        return "Ok";
    }
}
```

Ce code produit des sorties de journal similaires à ce qui suit :

Example format des journaux

```
{
    "timestamp": "2023-11-15T16:56:00.815Z",
    "level": "INFO",
```

```
    "message": "Handling s3 event",
    "logger": "example.HandlerS3",
    "AWSRequestId": "0bced576-3936-4e5a-9dcd-db9477b77f97",
    "awsRegion": "eu-south-1",
    "bucket": "java-logging-test-input-bucket",
    "eventName": "ObjectCreated:Put",
    "key": "test-folder/"
}
```

La configuration de construction prend en compte les dépendances d'exécution sur l'appendice Lambda et l'adaptateur SLF4 J, ainsi que les dépendances d'implémentation sur Log4j2.

Example build.gradle – Dépendances de journalisation

```
dependencies {
    ...
    'com.amazonaws:aws-lambda-java-log4j2:[1.6.0,)',
    'com.amazonaws:aws-lambda-java-events:[3.11.3,)',
    'org.apache.logging.log4j:log4j-layout-template-json:[2.17.1,)',
    'org.apache.logging.log4j:log4j-slf4j2-impl:[2.19.0,)',
    ...
}
```

Lorsque vous exécutez votre code localement pour des tests, l'objet contecte avec l'enregistreur Lambda n'est pas disponible, et il n'y a pas d'ID de requête que l'appender Lambda puisse utiliser. Pour des exemples de configurations de test, consultez les exemples d'applications dans la section suivante.

Utilisation d'autres outils et bibliothèques de journalisation

Powertools for AWS Lambda (Java) est une boîte à outils destinée aux développeurs qui permet de mettre en œuvre les meilleures pratiques sans serveur et d'accroître la rapidité des développeurs. L'utilitaire Logging (français non garanti) fournit un enregistreur optimisé pour Lambda qui inclut des informations supplémentaires sur le contexte de fonction pour toutes vos fonctions avec une sortie structurée en JSON. Utilisez cet utilitaire pour effectuer les opérations suivantes :

- Capturer les champs clés du contexte Lambda, démarrer à froid et structurer la sortie de la journalisation sous forme de JSON
- Journaliser les événements d'invocation Lambda lorsque cela est demandé (désactivé par défaut)

- Imprimer tous les journaux uniquement pour un pourcentage d'invocations via l'échantillonnage des journaux (désactivé par défaut)
- Ajouter des clés supplémentaires au journal structuré à tout moment
- Utiliser un formateur de journaux personnalisé (Apportez votre propre formateur) pour produire des journaux dans une structure compatible avec le RFC de journalisation de votre organisation

Utilisation de Powertools pour AWS Lambda (Java) et AWS SAM pour la journalisation structurée

Suivez les étapes ci-dessous pour télécharger, créer et déployer un exemple d'application Java Hello World avec des modules Powertools for AWS Lambda (Java) intégrés à l'aide du AWS SAM. Cette application met en œuvre un backend API de base et utilise Powertools pour émettre des journaux, des métriques et des traces. Elle se compose d'un point de terminaison Amazon API Gateway et d'une fonction Lambda. Lorsque vous envoyez une requête GET au point de terminaison API Gateway, la fonction Lambda appelle, envoie des journaux et des métriques au format métrique intégré à CloudWatch, et envoie des traces à. AWS X-Ray La fonction renvoie un message `hello world`.

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- Java 11
- AWS CLI version 2
- AWS SAM CLI version 1.75 ou ultérieure. Si vous disposez d'une ancienne version de la AWS SAM CLI, consultez la section Mise à niveau de la AWS SAM CLI.

Déployer un exemple d' AWS SAM application

1. Initialisez l'application à l'aide du modèle Hello World Java.

```
sam init --app-template hello-world-powertools-java --name sam-app --package-type
 Zip --runtime java11 --no-tracing
```

2. Créez l'application.

```
cd sam-app && sam build
```

3. Déployez l'application.

```
sam deploy --guided
```

4. Suivez les invites à l'écran. Appuyez sur `Enter` pour accepter les options par défaut fournies dans l'expérience interactive.

> **Note**
>
> Car l'autorisation n'a HelloWorldFunction peut-être pas été définie, est-ce que ça va ? , assurez-vous de participery.

5. Obtenez l'URL de l'application déployée :

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoquez le point de terminaison de l'API :

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

En cas de succès, vous obtiendrez cette réponse :

```
{"message":"hello world"}
```

7. Pour obtenir les journaux de la fonction, exécutez sam logs. Pour en savoir plus, consultez Utilisation des journaux dans le Guide du développeur AWS Serverless Application Model .

```
sam logs --stack-name sam-app
```

La sortie du journal se présente comme suit :

```
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.095000
 INIT_START Runtime Version: java:11.v15    Runtime Version ARN: arn:aws:lambda:eu-
central-1::runtime:0a25e3e7a1cc9ce404bc435eeb2ad358d8fa64338e618d0c224fe509403583ca
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.114000
 Picked up JAVA_TOOL_OPTIONS: -XX:+TieredCompilation -XX:TieredStopAtLevel=1
```

```
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.793000
 Transforming org/apache/logging/log4j/core/lookup/JndiLookup
 (lambdainternal.CustomerClassLoader@1a6c5a9e)
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:35.252000
 START RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765 Version: $LATEST
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.531000 {
  "_aws": {
    "Timestamp": 1675416276051,
    "CloudWatchMetrics": [
      {
        "Namespace": "sam-app-powerools-java",
        "Metrics": [
          {
            "Name": "ColdStart",
            "Unit": "Count"
          }
        ],
        "Dimensions": [
          [
            "Service",
            "FunctionName"
          ]
        ]
      }
    ]
  },
  "function_request_id": "7fcf1548-d2d4-41cd-a9a8-6ae47c51f765",
  "traceId":
 "Root=1-63dcd2d1-25f90b9d1c753a783547f4dd;Parent=e29684c1be352ce4;Sampled=1",
  "FunctionName": "sam-app-HelloWorldFunction-y9Iu1FLJJBGD",
  "functionVersion": "$LATEST",
  "ColdStart": 1.0,
  "Service": "service_undefined",
  "logStreamId": "2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81",
  "executionEnvironment": "AWS_Lambda_java11"
}
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.974000 Feb
 03, 2023 9:24:36 AM com.amazonaws.xray.AWSXRayRecorder <init>
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.993000 Feb
 03, 2023 9:24:36 AM com.amazonaws.xray.config.DaemonConfiguration <init>
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.993000
 INFO: Environment variable AWS_XRAY_DAEMON_ADDRESS is set. Emitting to daemon on
 address XXXX.XXXX.XXXX.XXXX:2000.
```

```
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:37.331000
 09:24:37.294 [main] INFO  helloworld.App - {"version":null,"resource":"/
hello","path":"/hello/","httpMethod":"GET","headers":{"Accept":"*/
*","CloudFront-Forwarded-Proto":"https","CloudFront-Is-Desktop-
Viewer":"true","CloudFront-Is-Mobile-Viewer":"false","CloudFront-Is-
SmartTV-Viewer":"false","CloudFront-Is-Tablet-Viewer":"false","CloudFront-
Viewer-ASN":"16509","CloudFront-Viewer-Country":"IE","Host":"XXXX.execute-
api.eu-central-1.amazonaws.com","User-Agent":"curl/7.86.0","Via":"2.0
 f0300a9921a99446a44423d996042050.cloudfront.net (CloudFront)","X-Amz-
Cf-Id":"t9W5ByT11HaY33NM8YioKECn_4eMpNsOMPfEVRczD7T1RdhbtiwV1Q==","X-
Amzn-Trace-Id":"Root=1-63dcd2d1-25f90b9d1c753a783547f4dd","X-Forwarded-
For":"XX.XXX.XXX.XX, XX.XXX.XXX.XX","X-Forwarded-Port":"443","X-
Forwarded-Proto":"https"},"multiValueHeaders":{"Accept":["*/
*"],"CloudFront-Forwarded-Proto":["https"],"CloudFront-Is-Desktop-Viewer":
["true"],"CloudFront-Is-Mobile-Viewer":["false"],"CloudFront-Is-SmartTV-
Viewer":["false"],"CloudFront-Is-Tablet-Viewer":["false"],"CloudFront-Viewer-
ASN":["16509"],"CloudFront-Viewer-Country":["IE"],"Host":["XXXX.execute-
api.eu-central-1.amazonaws.com"],"User-Agent":["curl/7.86.0"],"Via":["2.0
 f0300a9921a99446a44423d996042050.cloudfront.net (CloudFront)"],"X-Amz-
Cf-Id":["t9W5ByT11HaY33NM8YioKECn_4eMpNsOMPfEVRczD7T1RdhbtiwV1Q=="],"X-
Amzn-Trace-Id":["Root=1-63dcd2d1-25f90b9d1c753a783547f4dd"],"X-Forwarded-
For":["XXX, XXX"],"X-Forwarded-Port":["443"],"X-Forwarded-Proto":
["https"]},"queryStringParameters":null,"multiValueQueryStringParameters":null,"pathParamet
{"accountId":"XXX","stage":"Prod","resourceId":"at73a1","requestId":"ba09ecd2-
acf3-40f6-89af-fad32df67597","operationName":null,"identity":
{"cognitoIdentityPoolId":null,"accountId":null,"cognitoIdentityId":null,"caller":null,"apiK
hello","httpMethod":"GET","apiId":"XXX","path":"/Prod/
hello/","authorizer":null},"body":null,"isBase64Encoded":false}
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:37.351000
 09:24:37.351 [main] INFO  helloworld.App - Retrieving https://
checkip.amazonaws.com
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:39.313000 {
  "function_request_id": "7fcf1548-d2d4-41cd-a9a8-6ae47c51f765",
  "traceId":
 "Root=1-63dcd2d1-25f90b9d1c753a783547f4dd;Parent=e29684c1be352ce4;Sampled=1",
  "xray_trace_id": "1-63dcd2d1-25f90b9d1c753a783547f4dd",
  "functionVersion": "$LATEST",
  "Service": "service_undefined",
  "logStreamId": "2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81",
  "executionEnvironment": "AWS_Lambda_java11"
}
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:39.371000 END
 RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765
```

```
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:39.371000
 REPORT RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765    Duration: 4118.98 ms
 Billed Duration: 5275 ms    Memory Size: 512 MB    Max Memory Used: 152 MB    Init
 Duration: 1155.47 ms
XRAY TraceId: 1-63dcd2d1-25f90b9d1c753a783547f4dd    SegmentId: 3a028fee19b895cb
 Sampled: true
```

8. Il s'agit d'un point de terminaison d'API public accessible par Internet. Nous vous recommandons de supprimer le point de terminaison après un test.

```
sam delete
```

Gestion de la conservation des journaux

Les groupes de journaux ne sont pas supprimés automatiquement quand vous supprimez une fonction. Pour éviter de stocker les journaux indéfiniment, supprimez le groupe de journaux ou configurez une période de conservation après laquelle les journaux CloudWatch sont automatiquement supprimés. Pour configurer la conservation des journaux, ajoutez ce qui suit à votre AWS SAM modèle :

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Affichage des journaux dans la console Lambda

Vous pouvez utiliser la console Lambda pour afficher la sortie du journal après avoir invoqué une fonction Lambda.

Si votre code peut être testé à partir de l'éditeur Code intégré, vous trouverez les journaux dans les résultats d'exécution. Lorsque vous utilisez la fonctionnalité de test de console pour invoquer une fonction, vous trouverez Sortie du journal dans la section Détails.

Afficher les journaux dans la CloudWatch console

Vous pouvez utiliser la CloudWatch console Amazon pour consulter les journaux de toutes les invocations de fonctions Lambda.

Pour afficher les journaux sur la CloudWatch console

1. Ouvrez la page Groupes de journaux sur la CloudWatch console.
2. Choisissez le groupe de journaux pour votre fonction (***your-function-name***/aws/lambda/).
3. Choisissez un flux de journaux.

Chaque flux de journal correspond à une instance de votre fonction. Un flux de journal apparaît lorsque vous mettez à jour votre fonction Lambda et lorsque des instances supplémentaires sont créées pour gérer les appels simultanés. Pour trouver les journaux d'un appel spécifique, nous vous recommandons d'instrumenter votre fonction avec. AWS X-Ray X-Ray enregistre des détails sur la demande et le flux de journaux dans le suivi.

Afficher les journaux à l'aide de AWS Command Line Interface (AWS CLI)

AWS CLI Il s'agit d'un outil open source qui vous permet d'interagir avec les AWS services à l'aide de commandes dans votre interface de ligne de commande. Pour effectuer les étapes de cette section, vous devez disposer de la version 2 de l'AWS CLI.

Vous pouvez utiliser AWS CLI pour récupérer les journaux d'une invocation à l'aide de l'option de commande `--log-type`. La réponse inclut un champ `LogResult` qui contient jusqu'à 4 Ko de journaux codés en base64 provenant de l'invocation.

Example récupérer un ID de journal

L'exemple suivant montre comment récupérer un ID de journal à partir du champ `LogResult` d'une fonction nommée `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Vous devriez voir la sortie suivante:

```
{
    "StatusCode": 200,
```

```
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example décoder les journaux

Dans la même invite de commandes, utilisez l'utilitaire `base64` pour décoder les journaux. L'exemple suivant montre comment récupérer les journaux encodés en base64 pour `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

Vous devriez voir la sortie suivante :

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

L'utilitaire `base64` est disponible sous Linux, macOS et Ubuntu sous Windows. Les utilisateurs de macOS auront peut-être besoin d'utiliser `base64 -D`.

Example Script get-logs.sh

Dans la même invite de commandes, utilisez le script suivant pour télécharger les cinq derniers événements de journalisation. Le script utilise `sed` pour supprimer les guillemets du fichier de sortie et attend 15 secondes pour permettre la mise à disposition des journaux. La sortie comprend la réponse de Lambda, ainsi que la sortie de la commande `get-log-events`.

Copiez le contenu de l'exemple de code suivant et enregistrez-le dans votre répertoire de projet Lambda sous `get-logs.sh`.

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS et Linux (uniquement)

Dans la même invite de commandes, les utilisateurs macOS et Linux peuvent avoir besoin d'exécuter la commande suivante pour s'assurer que le script est exécutable.

```
chmod -R 755 get-logs.sh
```

Example récupérer les cinq derniers événements de journal

Dans la même invite de commande, exécutez le script suivant pour obtenir les cinq derniers événements de journalisation.

```
./get-logs.sh
```

Vous devriez voir la sortie suivante:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
```

```
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


Suppression de journaux

Les groupes de journaux ne sont pas supprimés automatiquement quand vous supprimez une fonction. Pour éviter de stocker des journaux indéfiniment, supprimez le groupe de journaux ou configurez une période de conservation à l'issue de laquelle les journaux sont supprimés automatiquement.

Exemple de code de journalisation

Le GitHub référentiel de ce guide inclut des exemples d'applications illustrant l'utilisation de différentes configurations de journalisation. Chaque exemple d'application inclut des scripts pour faciliter le déploiement et le nettoyage, un AWS SAM modèle et des ressources de support.

Exemples d’applications Lambda en Java

- example-java : fonction Java qui montre comment utiliser Lambda pour traiter les commandes. Cette fonction montre comment définir et désérialiser un objet d'événement d'entrée personnalisé, utiliser le AWS SDK et enregistrer les sorties.
- java-basic – Ensemble de fonctions Java minimales avec des tests unitaires et une configuration de journalisation variable.
- java events – Ensemble de fonctions Java contenant du code squelette permettant de gérer les événements de divers services tels qu’Amazon API Gateway, Amazon SQS et Amazon Kinesis. Ces fonctions utilisent la dernière version de la aws-lambda-java-eventsbibliothèque (3.0.0 et versions ultérieures). Ces exemples ne nécessitent pas le AWS SDK comme dépendance.
- s3-java – Fonction Java qui traite les événements de notification d’Amazon S3 et utilise la bibliothèque de classes Java (JCL) pour créer des miniatures à partir de fichiers d’image chargés.
- layer-java : fonction Java qui illustre comment utiliser une couche Lambda pour empaqueter les dépendances séparément du code de votre fonction principale.

L’exemple `java-basic` d’application présente une configuration de journalisation minimale qui prend en charge les tests de journalisation. Le code du gestionnaire utilise l’enregistreur `LambdaLogger` fourni par l’objet de contexte. Pour les tests, l’application utilise une classe `TestLogger` personnalisée qui implémente l’interface `LambdaLogger` avec un logger Log4j2. Il utilise SLF4 J comme façade pour assurer la compatibilité avec le AWS SDK. Les bibliothèques de journalisation sont exclues de la sortie de build pour limiter la taille du package de déploiement.

Instrumentation du code Java dans AWS Lambda

Lambda s'intègre pour vous aider AWS X-Ray à suivre, à déboguer et à optimiser les applications Lambda. Vous pouvez utiliser X-Ray pour suivre une demande lorsque celle-ci parcourt les ressources de votre application, qui peuvent inclure des fonctions Lambda et d'autres services AWS .

Pour envoyer des données de suivi à X-Ray, vous pouvez utiliser l'une des deux bibliothèques SDK suivantes :

- AWS Distro for OpenTelemetry (ADOT) — Une distribution sécurisée, prête pour la production et AWS prise en charge du SDK (). OpenTelemetry OTel
- Kit SDK AWS X-Ray pour Java – Un kit SDK permettant de générer et d'envoyer des données de suivi à X-Ray.
- Powertools for AWS Lambda (Java) — Une boîte à outils pour les développeurs permettant de mettre en œuvre les meilleures pratiques sans serveur et d'accroître la rapidité des développeurs.

Chacune d'entre elles SDKs propose des moyens d'envoyer vos données de télémétrie au service X-Ray. Vous pouvez ensuite utiliser X-Ray pour afficher, filtrer et avoir un aperçu des métriques de performance de votre application, afin d'identifier les problèmes et les occasions d'optimiser votre application.

> **Important**
>
> Les outils X-Ray et Powertools pour AWS Lambda SDKs font partie d'une solution d'instrumentation étroitement intégrée proposée par AWS. Les couches ADOT Lambda font partie d'une norme industrielle pour l'instrumentation de traçage qui collecte plus de données en général, mais qui peut ne pas convenir à tous les cas d'utilisation. Vous pouvez implémenter le end-to-end traçage dans X-Ray en utilisant l'une ou l'autre des solutions. Pour en savoir plus sur le choix entre les deux, consultez Choosing between the AWS Distro for Open Telemetry and X-Ray. SDKs

Sections

- Utilisation de Powertools pour AWS Lambda (Java) et AWS SAM pour le traçage
- Utilisation de Powertools pour AWS Lambda (Java) et AWS CDK pour le traçage
- Utilisation d'ADOT pour instrumenter vos fonctions Java

- Utilisation du kit SDK X-Ray pour instrumenter vos fonctions Java
- Activation du suivi avec la console Lambda
- Activation du suivi avec l'API Lambda
- Activation du traçage avec CloudFormation
- Interprétation d'un suivi X-Ray
- Stockage des dépendances d'exécution dans une couche (kit SDK X-Ray)
- Suivi de X-Ray dans des exemples d'applications (kit SDK X-Ray)

Utilisation de Powertools pour AWS Lambda (Java) et AWS SAM pour le traçage

Suivez les étapes ci-dessous pour télécharger, créer et déployer un exemple d'application Java Hello World avec des modules Powertools for AWS Lambda (Java) intégrés à l'aide du AWS SAM. Cette application met en œuvre un backend API de base et utilise Powertools pour émettre des journaux, des métriques et des traces. Elle se compose d'un point de terminaison Amazon API Gateway et d'une fonction Lambda. Lorsque vous envoyez une requête GET au point de terminaison API Gateway, la fonction Lambda appelle, envoie des journaux et des métriques au format métrique intégré à CloudWatch, et envoie des traces à. AWS X-Ray La fonction renvoie un message `hello world`.

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- Java 11 ou version ultérieure
- AWS CLI version 2
- AWS SAM CLI version 1.75 ou ultérieure. Si vous disposez d'une ancienne version de la AWS SAM CLI, consultez la section Mise à niveau de la AWS SAM CLI.

Déployer un exemple d' AWS SAM application

1. Initialisez l'application à l'aide du modèle Hello World Java.

```
sam init --app-template hello-world-powertools-java --name sam-app --package-type
 Zip --runtime java11 --no-tracing
```

2. Créez l'application.

```
cd sam-app && sam build
```

3. Déployez l'application.

```
sam deploy --guided
```

4. Suivez les invites à l'écran. Appuyez sur `Enter` pour accepter les options par défaut fournies dans l'expérience interactive.

> **Note**
>
> Car l'autorisation n'a HelloWorldFunction peut-être pas été définie, est-ce que ça va ? , assurez-vous de participery.

5. Obtenez l'URL de l'application déployée :

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoquez le point de terminaison de l'API :

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

En cas de succès, vous obtiendrez cette réponse :

```
{"message":"hello world"}
```

7. Pour obtenir les traces de la fonction, exécutez sam traces.

```
sam traces
```

La sortie de la trace ressemble à ceci :

```
New XRay Service Graph
```

```
  Start time: 2025-02-03 14:31:48+01:00
  End time: 2025-02-03 14:31:48+01:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-y9Iu1FLJJBGD -
 Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 5.587
  Reference Id: 1 - client - sam-app-HelloWorldFunction-y9Iu1FLJJBGD - Edges: [0]
   Summary_statistics:
     - total requests: 0
     - ok count(2XX): 0
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0

XRay Event [revision 3] at (2025-02-03T14:31:48.500000) with id
 (1-63dd0cc4-3c869dec72a586875da39777) and duration (5.603s)
 - 5.587s - sam-app-HelloWorldFunction-y9Iu1FLJJBGD [HTTP: 200]
 - 4.053s - sam-app-HelloWorldFunction-y9Iu1FLJJBGD
   - 1.181s - Initialization
   - 4.037s - Invocation
     - 1.981s - ## handleRequest
       - 1.840s - ## getPageContents
   - 0.000s - Overhead
```

8. Il s’agit d’un point de terminaison d’API public accessible par Internet. Nous vous recommandons de supprimer le point de terminaison après un test.

```
sam delete
```

Utilisation de Powertools pour AWS Lambda (Java) et AWS CDK pour le traçage

Suivez les étapes ci-dessous pour télécharger, créer et déployer un exemple d'application Java Hello World avec des modules Powertools for AWS Lambda (Java) intégrés à l'aide du AWS CDK. Cette application met en œuvre un backend API de base et utilise Powertools pour émettre des journaux, des métriques et des traces. Elle se compose d’un point de terminaison Amazon API

Gateway et d'une fonction Lambda. Lorsque vous envoyez une requête GET au point de terminaison API Gateway, la fonction Lambda appelle, envoie des journaux et des métriques au format métrique intégré à CloudWatch, et envoie des traces à. AWS X-Ray La fonction renvoie un message hello world.

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- Java 11 ou version ultérieure
- AWS CLI version 2
- AWS CDK version 2
- AWS SAM CLI version 1.75 ou ultérieure. Si vous disposez d'une ancienne version de la AWS SAM CLI, consultez la section Mise à niveau de la AWS SAM CLI.

Déployer un exemple d' AWS CDK application

1. Créez un répertoire de projets pour votre nouvelle application.

```
mkdir hello-world
cd hello-world
```

2. Initialisez l'application.

```
cdk init app --language java
```

3. Créez un projet Maven avec la commande suivante :

```
mkdir app
cd app
mvn archetype:generate -DgroupId=helloworld -DartifactId=Function -
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

4. Ouvrez `pom.xml` dans le répertoire `hello-world\app\Function` et remplacez le code existant par le code suivant qui inclut les dépendances et les plugins Maven pour Powertools.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
```

```
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>helloworld</groupId>
  <artifactId>Function</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Function</name>
  <url>http://maven.apache.org</url>
<properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
    <log4j.version>2.17.2</log4j.version>
</properties>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>software.amazon.lambda</groupId>
            <artifactId>powertools-tracing</artifactId>
            <version>1.3.0</version>
        </dependency>
        <dependency>
            <groupId>software.amazon.lambda</groupId>
            <artifactId>powertools-metrics</artifactId>
            <version>1.3.0</version>
        </dependency>
        <dependency>
            <groupId>software.amazon.lambda</groupId>
            <artifactId>powertools-logging</artifactId>
            <version>1.3.0</version>
        </dependency>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-core</artifactId>
            <version>1.2.2</version>
        </dependency>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-events</artifactId>
```

```
            <version>3.11.1</version>
        </dependency>
  </dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>aspectj-maven-plugin</artifactId>
            <version>1.14.0</version>
            <configuration>
                <source>${maven.compiler.source}</source>
                <target>${maven.compiler.target}</target>
                <complianceLevel>${maven.compiler.target}</complianceLevel>
                <aspectLibraries>
                    <aspectLibrary>
                        <groupId>software.amazon.lambda</groupId>
                        <artifactId>powertools-tracing</artifactId>
                    </aspectLibrary>
                    <aspectLibrary>
                        <groupId>software.amazon.lambda</groupId>
                        <artifactId>powertools-metrics</artifactId>
                    </aspectLibrary>
                    <aspectLibrary>
                        <groupId>software.amazon.lambda</groupId>
                        <artifactId>powertools-logging</artifactId>
                    </aspectLibrary>
                </aspectLibraries>
            </configuration>
            <executions>
                <execution>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>3.4.1</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
```

```
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            <transformers>
                                <transformer

 implementation="com.github.edwgiz.maven_shade_plugin.log4j2_cache_transformer.PluginsCache
                                </transformer>
                            </transformers>
                            <createDependencyReducedPom>false</
createDependencyReducedPom>
                            <finalName>function</finalName>

                        </configuration>
                    </execution>
                </executions>
                <dependencies>
                    <dependency>
                        <groupId>com.github.edwgiz</groupId>
                        <artifactId>maven-shade-plugin.log4j2-cachefile-
transformer</artifactId>
                        <version>2.15</version>
                    </dependency>
                </dependencies>
        </plugin>
    </plugins>
</build>
</project>
```

5. Créez le répertoire `hello-world\app\src\main\resource` et créez `log4j.xml` pour la configuration du journal.

```
mkdir -p src/main/resource
cd src/main/resource
touch log4j.xml
```

6. Ouvrez `log4j.xml` et ajoutez le code suivant.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
    <Appenders>
        <Console name="JsonAppender" target="SYSTEM_OUT">
```

```
            <JsonTemplateLayout
 eventTemplateUri="classpath:LambdaJsonLayout.json" />
        </Console>
    </Appenders>
    <Loggers>
        <Logger name="JsonLogger" level="INFO" additivity="false">
            <AppenderRef ref="JsonAppender"/>
        </Logger>
        <Root level="info">
            <AppenderRef ref="JsonAppender"/>
        </Root>
    </Loggers>
</Configuration>
```

7. Ouvrez `App.java` à partir du répertoire `hello-world\app\Function\src\main\java\helloworld` et remplacez le code existant par le code suivant. Il s'agit du code de la fonction Lambda.

```
package helloworld;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;
import java.util.stream.Collectors;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import software.amazon.lambda.powertools.logging.Logging;
import software.amazon.lambda.powertools.metrics.Metrics;
import software.amazon.lambda.powertools.tracing.CaptureMode;
import software.amazon.lambda.powertools.tracing.Tracing;

import static software.amazon.lambda.powertools.tracing.CaptureMode.*;

/**
 * Handler for requests to Lambda function.
```

```
 */
public class App implements RequestHandler<APIGatewayProxyRequestEvent,
 APIGatewayProxyResponseEvent> {
    Logger log = LogManager.getLogger(App.class);


    @Logging(logEvent = true)
    @Tracing(captureMode = DISABLED)
    @Metrics(captureColdStart = true)
    public APIGatewayProxyResponseEvent handleRequest(final
 APIGatewayProxyRequestEvent input, final Context context) {
        Map<String, String> headers = new HashMap<>();
        headers.put("Content-Type", "application/json");
        headers.put("X-Custom-Header", "application/json");

        APIGatewayProxyResponseEvent response = new APIGatewayProxyResponseEvent()
                .withHeaders(headers);
        try {
            final String pageContents = this.getPageContents("https://
checkip.amazonaws.com");
            String output = String.format("{ \"message\": \"hello world\",
 \"location\": \"%s\" }", pageContents);

            return response
                    .withStatusCode(200)
                    .withBody(output);
        } catch (IOException e) {
            return response
                    .withBody("{}")
                    .withStatusCode(500);
        }
    }
    @Tracing(namespace = "getPageContents")
    private String getPageContents(String address) throws IOException {
        log.info("Retrieving {}", address);
        URL url = new URL(address);
        try (BufferedReader br = new BufferedReader(new
 InputStreamReader(url.openStream()))) {
            return br.lines().collect(Collectors.joining(System.lineSeparator()));
        }
    }
}
```

8. Ouvrez `HelloWorldStack.java` à partir du répertoire `hello-world\src\main\java\com\myorg` et remplacez le code existant par le code suivant. Ce code utilise le constructeur Lambda et le constructeur ApiGatewayv 2 pour créer une API REST et une fonction Lambda.

```
package com.myorg;

import software.amazon.awscdk.*;
import software.amazon.awscdk.services.apigatewayv2.alpha.*;
import
 software.amazon.awscdk.services.apigatewayv2.integrations.alpha.HttpLambdaIntegration;
import
 software.amazon.awscdk.services.apigatewayv2.integrations.alpha.HttpLambdaIntegrationProps
import software.amazon.awscdk.services.lambda.Code;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.FunctionProps;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.lambda.Tracing;
import software.amazon.awscdk.services.logs.RetentionDays;
import software.amazon.awscdk.services.s3.assets.AssetOptions;
import software.constructs.Construct;

import java.util.Arrays;
import java.util.List;

import static java.util.Collections.singletonList;
import static software.amazon.awscdk.BundlingOutput.ARCHIVED;

public class HelloWorldStack extends Stack {
    public HelloWorldStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public HelloWorldStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        List<String> functionPackagingInstructions = Arrays.asList(
                "/bin/sh",
                "-c",
                "cd Function " +
                        "&& mvn clean install " +
                        "&& cp /asset-input/Function/target/function.jar /asset-
output/"
```

```
        );
        BundlingOptions.Builder builderOptions = BundlingOptions.builder()
                .command(functionPackagingInstructions)
                .image(Runtime.JAVA_11.getBundlingImage())
                .volumes(singletonList(
                        // Mount local .m2 repo to avoid download all the
dependencies again inside the container
                        DockerVolume.builder()
                                .hostPath(System.getProperty("user.home") +
"/.m2/")
                                .containerPath("/root/.m2/")
                                .build()
                ))
                .user("root")
                .outputType(ARCHIVED);

        Function function = new Function(this, "Function", FunctionProps.builder()
                .runtime(Runtime.JAVA_11)
                .code(Code.fromAsset("app", AssetOptions.builder()
                        .bundling(builderOptions
                                .command(functionPackagingInstructions)
                                .build())
                        .build()))
                .handler("helloworld.App::handleRequest")
                .memorySize(1024)
                .tracing(Tracing.ACTIVE)
                .timeout(Duration.seconds(10))
                .logRetention(RetentionDays.ONE_WEEK)
                .build());

        HttpApi httpApi = new HttpApi(this, "sample-api", HttpApiProps.builder()
                .apiName("sample-api")
                .build());

        httpApi.addRoutes(AddRoutesOptions.builder()
                .path("/")
                .methods(singletonList(HttpMethod.GET))
                .integration(new HttpLambdaIntegration("function", function,
HttpLambdaIntegrationProps.builder()
                        .payloadFormatVersion(PayloadFormatVersion.VERSION_2_0)
                        .build()))
                .build());

        new CfnOutput(this, "HttpApi", CfnOutputProps.builder()
```

```
                .description("Url for Http Api")
                .value(httpApi.getApiEndpoint())
                .build());
    }
}
```

9. Ouvrez `pom.xml` à partir du répertoire `hello-world` et remplacez le code existant par le code suivant.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd"
         xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.myorg</groupId>
    <artifactId>hello-world</artifactId>
    <version>0.1</version>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <cdk.version>2.70.0</cdk.version>
        <constructs.version>[10.0.0,11.0.0)</constructs.version>
        <junit.version>5.7.1</junit.version>
    </properties>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>

            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
                <version>3.0.0</version>
```

```
                <configuration>
                    <mainClass>com.myorg.HelloWorldApp</mainClass>
                </configuration>
            </plugin>
        </plugins>
    </build>

    <dependencies>
        <!-- AWS Cloud Development Kit -->
        <dependency>
            <groupId>software.amazon.awscdk</groupId>
            <artifactId>aws-cdk-lib</artifactId>
            <version>${cdk.version}</version>
        </dependency>
        <dependency>
            <groupId>software.constructs</groupId>
            <artifactId>constructs</artifactId>
            <version>${constructs.version}</version>
        </dependency>
        <dependency>
          <groupId>org.junit.jupiter</groupId>
          <artifactId>junit-jupiter</artifactId>
          <version>${junit.version}</version>
          <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>software.amazon.awscdk</groupId>
            <artifactId>apigatewayv2-alpha</artifactId>
            <version>${cdk.version}-alpha.0</version>
        </dependency>
        <dependency>
            <groupId>software.amazon.awscdk</groupId>
            <artifactId>apigatewayv2-integrations-alpha</artifactId>
            <version>${cdk.version}-alpha.0</version>
        </dependency>
    </dependencies>
</project>
```

10. Assurez-vous d'être dans le répertoire `hello-world` et déployez votre application.

```
cdk deploy
```

11. Obtenez l'URL de l'application déployée :

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`HttpApi`].OutputValue' --output text
```

12. Invoquez le point de terminaison de l'API :

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

En cas de succès, vous obtiendrez cette réponse :

```
{"message":"hello world"}
```

13. Pour obtenir les traces de la fonction, exécutez sam traces.

```
sam traces
```

La sortie de la trace ressemble à ceci :

```
New XRay Service Graph
  Start time: 2025-02-03 14:59:50+00:00
  End time: 2025-02-03 14:59:50+00:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-YBg8yfYtOc9j -
 Edges: [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.924
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-YBg8yfYtOc9j
 - Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.016
  Reference Id: 2 - client - sam-app-HelloWorldFunction-YBg8yfYtOc9j - Edges: [0]
   Summary_statistics:
     - total requests: 0
     - ok count(2XX): 0
     - error count(4XX): 0
```

```
     - fault count(5XX): 0
     - total response time: 0

XRay Event [revision 1] at (2025-02-03T14:59:50.204000) with id
 (1-63dd2166-434a12c22e1307ff2114f299) and duration (0.924s)
 - 0.924s - sam-app-HelloWorldFunction-YBg8yfYtOc9j [HTTP: 200]
 - 0.016s - sam-app-HelloWorldFunction-YBg8yfYtOc9j
   - 0.739s - Initialization
   - 0.016s - Invocation
     - 0.013s - ## lambda_handler
       - 0.000s - ## app.hello
   - 0.000s - Overhead
```

14. Il s'agit d'un point de terminaison d'API public accessible par Internet. Nous vous recommandons de supprimer le point de terminaison après un test.

```
cdk destroy
```

Utilisation d'ADOT pour instrumenter vos fonctions Java

ADOT fournit des couches Lambda entièrement gérées qui regroupent tout ce dont vous avez besoin pour collecter des données de télémétrie à l'aide du SDK. OTel En consommant cette couche, vous pouvez instrumenter vos fonctions Lambda sans avoir à modifier le code de fonction. Vous pouvez également configurer votre couche pour effectuer une initialisation personnalisée de OTel. Pour de plus amples informations, veuillez consulter Configuration personnalisée pour ADOT Collector sur Lambda dans la documentation ADOT.

Pour les environnements d'exécution Java, vous pouvez choisir entre deux couches à consommer :

- AWS couche Lambda gérée pour ADOT Java (agent d'instrumentation automatique) : cette couche transforme automatiquement le code de votre fonction au démarrage pour collecter des données de suivi. Pour obtenir des instructions détaillées sur la façon d'utiliser cette couche avec l'agent Java ADOT, consultez AWS Distro for Lambda OpenTelemetry Support for Java (agent d'instrumentation automatique) dans la documentation ADOT.
- AWS couche Lambda gérée pour ADOT Java — Cette couche fournit également une instrumentation intégrée pour les fonctions Lambda, mais elle nécessite quelques modifications de code manuelles pour initialiser le SDK. OTel Pour obtenir des instructions détaillées sur la façon de consommer cette couche, consultez AWS Distro for OpenTelemetry Lambda Support for Java dans la documentation ADOT.

Utilisation du kit SDK X-Ray pour instrumenter vos fonctions Java

Pour enregistrer des données sur les appels effectués par votre fonction à d'autres ressources et services de votre application, vous pouvez ajouter le kit SDK X-Ray pour Java à la configuration de votre création. L'exemple suivant montre une configuration de build Gradle qui inclut les bibliothèques qui activent l'instrumentation automatique des AWS SDK for Java 2.x clients.

Example build.gradle – Dépendances du suivi

```
dependencies {
    implementation platform('software.amazon.awssdk:bom:2.16.1')
    implementation platform('com.amazonaws:aws-xray-recorder-sdk-bom:2.11.0')
    ...
    implementation 'com.amazonaws:aws-xray-recorder-sdk-core'
    implementation 'com.amazonaws:aws-xray-recorder-sdk-aws-sdk'
    implementation 'com.amazonaws:aws-xray-recorder-sdk-aws-sdk-v2-instrumentor'
    ...
}
```

Une fois que vous avez ajouté les bonnes dépendances et effectué les modifications de code nécessaires, activez le suivi dans la configuration de votre fonction via la console Lambda ou l'API.

Activation du suivi avec la console Lambda

Pour activer/désactiver le traçage actif sur votre fonction Lambda avec la console, procédez comme suit :

Pour activer le traçage actif

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Choisissez Configuration, puis choisissez Outils de surveillance et d’opérations.
4. Dans Outils de surveillance supplémentaires, choisissez Modifier.
5. Sous Signaux CloudWatch d'application et AWS X-Ray sélectionnez Activer les traces de service Lambda.
6. Choisissez Enregistrer.

Activation du suivi avec l'API Lambda

Configurez le suivi sur votre fonction Lambda avec le AWS SDK AWS CLI or, utilisez les opérations d'API suivantes :

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

L'exemple de AWS CLI commande suivant active le suivi actif sur une fonction nommée my-function.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Le mode de suivi fait partie de la configuration spécifique de la version lorsque vous publiez une version de votre fonction. Vous ne pouvez pas modifier le mode de suivi sur une version publiée.

Activation du traçage avec CloudFormation

Pour activer le suivi d'une `AWS::Lambda::Function` ressource dans un CloudFormation modèle, utilisez la `TracingConfig` propriété.

Example function-inline.yml – Configuration du suivi

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

Pour une `AWS::Serverless::Function` ressource AWS Serverless Application Model (AWS SAM), utilisez la `Tracing` propriété.

Example template.yml – Configuration du suivi

```
Resources:
```

```
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

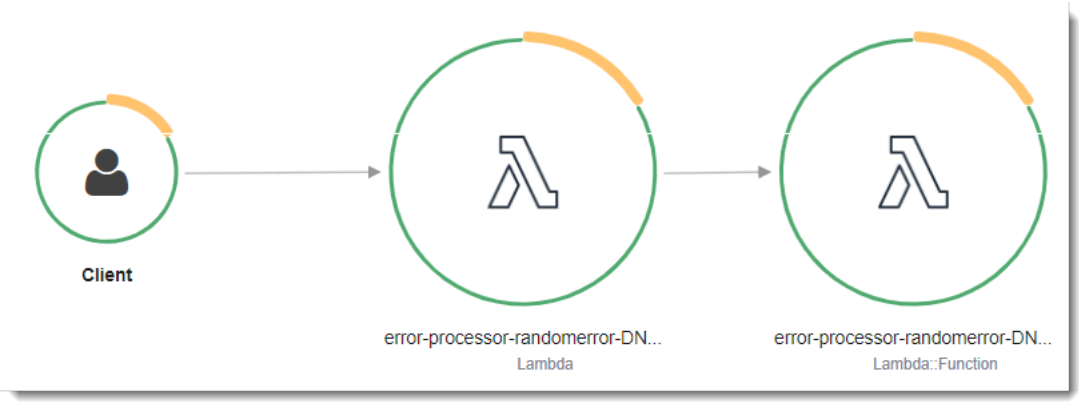
Interprétation d'un suivi X-Ray

Votre fonction a besoin d'une autorisation pour charger des données de suivi vers X-Ray. Lorsque vous activez le suivi actif dans la console Lambda, Lambda ajoute les autorisations requises au rôle d'exécution de votre fonction. Dans le cas contraire, ajoutez la AWSXRayDaemonWriteAccesspolitique au rôle d'exécution.

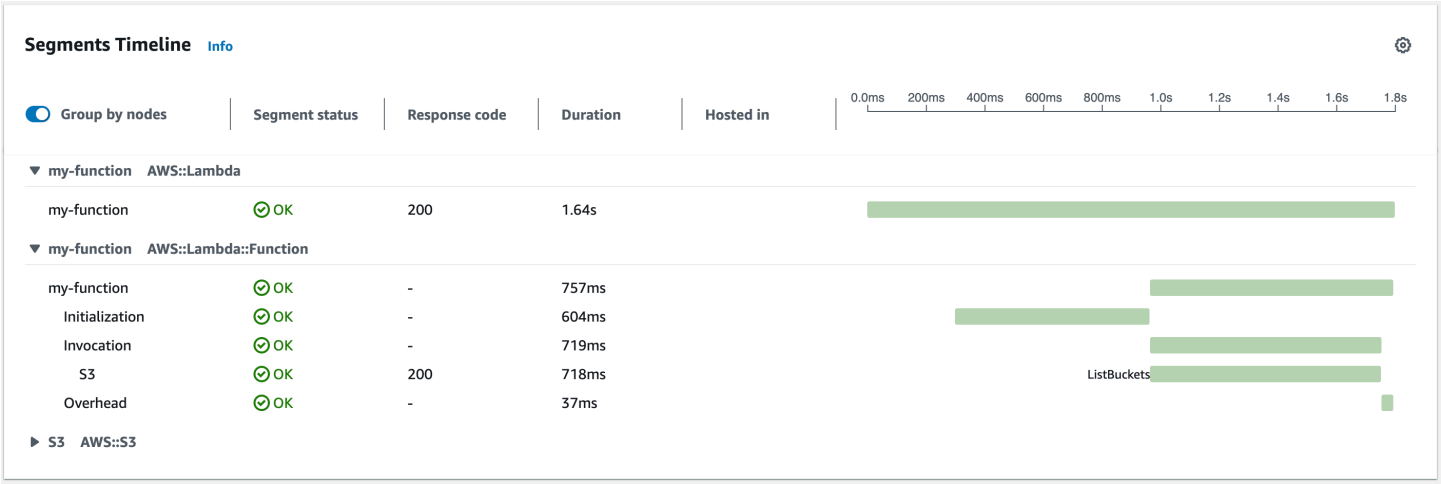
Une fois que vous avez configuré le suivi actif, vous pouvez observer des demandes spécifiques via votre application. Le graphique de services X-Ray affiche des informations sur votre application et tous ses composants. L'exemple suivant montre une application dotée de deux fonctions. La fonction principale traite les événements et renvoie parfois des erreurs. La deuxième fonction située en haut traite les erreurs qui apparaissent dans le groupe de journaux de la première et utilise le AWS SDK pour appeler X-Ray, Amazon Simple Storage Service (Amazon S3) et Amazon Logs. CloudWatch

X-Ray ne trace pas toutes les requêtes vers votre application. X-Ray applique un algorithme d'échantillonnage pour s'assurer que le suivi est efficace, tout en fournissant un échantillon représentatif de toutes les demandes. Le taux d'échantillonnage est 1 demande par seconde et 5 % de demandes supplémentaires. Vous ne pouvez pas configurer ce taux d'échantillonnage X-Ray pour vos fonctions.

Dans X-Ray, un suivi enregistre des informations sur une demande traitée par un ou plusieurs services. Lambda enregistre deux segments par suivi, ce qui a pour effet de créer deux nœuds sur le graphique du service. L'image suivante met en évidence ces deux nœuds :



Le premier nœud sur la gauche représente le service Lambda qui reçoit la demande d'invocation. Le deuxième nœud représente votre fonction Lambda spécifique. L'exemple suivant illustre une trace avec ces deux segments. Les deux sont nommés my-function, mais l'un a pour origine `AWS::Lambda` et l'autre a pour origine `AWS::Lambda::Function`. Si le segment `AWS::Lambda` affiche une erreur, cela signifie que le service Lambda a rencontré un problème. Si le segment `AWS::Lambda::Function` affiche une erreur, cela signifie que votre fonction a rencontré un problème.



Cet exemple développe le segment `AWS::Lambda::Function` pour afficher ses trois sous-segments.

> **Note**
>
> AWS met actuellement en œuvre des modifications du service Lambda. En raison de ces modifications, vous pouvez constater des différences mineures entre la structure et le contenu des messages du journal système et des segments de suivi émis par les différentes fonctions Lambda de votre Compte AWS.
> L'exemple de suivi présenté ici illustre le segment de fonction à l'ancienne. Les différences entre les segments à l'ancienne et de style moderne sont décrites dans les paragraphes suivants.
> Ces modifications seront mises en œuvre au cours des prochaines semaines, et toutes les fonctions, Régions AWS sauf en Chine et dans les GovCloud régions, passeront à l'utilisation du nouveau format des messages de journal et des segments de trace.

Le segment de fonction à l'ancienne contient les sous-segments suivants :

- Initialization (Initialisation) : représente le temps passé à charger votre fonction et à exécuter le code d'initialisation. Ce sous-segment apparaît pour le premier événement traité par chaque instance de votre fonction.
- Invocation – Représente le temps passé à exécuter votre code de gestionnaire.
- Overhead (Travail supplémentaire) – Représente le temps que le fichier d'exécution Lambda passe à se préparer à gérer l'événement suivant.

Le segment de fonction de style moderne ne contient pas de sous-segment `Invocation`. À la place, les sous-segments du client sont directement rattachés au segment de fonction. Pour plus d'informations sur la structure des segments de fonction à l'ancienne et de style moderne, consultez the section called "Comprendre les suivis X-Ray".

> **Note**
>
> Les fonctions Lambda SnapStart incluent également un sous-segment `Restore`. Le sous-segment `Restore` indique le temps nécessaire à Lambda pour restaurer un instantané, charger l'exécution et exécuter les éventuels hooks d'exécution après restauration. Le processus de restauration des instantanés peut inclure du temps consacré à des activités en dehors de la MicroVM. Cette heure est indiquée dans le sous-segment `Restore`. Le temps passé en dehors de la microVM pour restaurer un instantané ne vous est pas facturé.

Vous pouvez également utiliser des clients HTTP, enregistrer des requêtes SQL et créer des sous-segments personnalisés avec des annotations et des métadonnées. Pour plus d'informations, consultez Kit SDK AWS X-Ray pour Java dans le Guide du développeur AWS X-Ray .

> **Tarification**
>
> Vous pouvez utiliser le traçage X-Ray gratuitement chaque mois jusqu'à une certaine limite dans le cadre du niveau AWS gratuit. Au-delà de ce seuil, X-Ray facture le stockage et la récupération du suivi. Pour en savoir plus, consultez Pricing AWS X-Ray (Tarification).

Stockage des dépendances d'exécution dans une couche (kit SDK X-Ray)

Si vous utilisez le SDK X-Ray pour instrumenter le code de fonction des clients du AWS SDK, votre package de déploiement peut devenir très volumineux. Pour éviter de charger des dépendances d'environnement d'exécution chaque fois que vous mettez à jour votre code de fonction, empaquetez le kit SDK X-Ray dans une couche Lambda.

L'exemple suivant montre une ressource `AWS::Serverless::LayerVersion` qui stocke AWS SDK pour Java et le kit SDK X-Ray pour Java.

Example template.yml : couche de dépendances

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: build/distributions/blank-java.zip
      Tracing: Active
      Layers:
        - !Ref libs
      ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-java-lib
      Description: Dependencies for the blank-java sample app.
      ContentUri: build/blank-java-lib.zip
      CompatibleRuntimes:
        - java25
```

Avec cette configuration, vous ne mettez à jour les fichiers de couche de bibliothèque que si vous modifiez vos dépendances d'exécution. Étant donné que le package de déploiement de la fonction contient uniquement votre code, cela peut contribuer à réduire les temps de chargement.

La création d'une couche pour les dépendances nécessite des modifications de configuration de création pour créer l'archive des couches avant le déploiement. Pour un exemple pratique, consultez l'exemple d'application java-basic sur. GitHub

Suivi de X-Ray dans des exemples d'applications (kit SDK X-Ray)

Le GitHub référentiel de ce guide inclut des exemples d'applications illustrant l'utilisation du traçage par rayons X. Chaque exemple d'application inclut des scripts pour faciliter le déploiement et le nettoyage, un AWS SAM modèle et des ressources de support.

Exemples d'applications Lambda en Java

- example-java : fonction Java qui montre comment utiliser Lambda pour traiter les commandes. Cette fonction montre comment définir et désérialiser un objet d'événement d'entrée personnalisé, utiliser le AWS SDK et enregistrer les sorties.
- java-basic – Ensemble de fonctions Java minimales avec des tests unitaires et une configuration de journalisation variable.
- java events – Ensemble de fonctions Java contenant du code squelette permettant de gérer les événements de divers services tels qu'Amazon API Gateway, Amazon SQS et Amazon Kinesis. Ces fonctions utilisent la dernière version de la aws-lambda-java-eventsbibliothèque (3.0.0 et versions ultérieures). Ces exemples ne nécessitent pas le AWS SDK en tant que dépendance.
- s3-java – Fonction Java qui traite les événements de notification d'Amazon S3 et utilise la bibliothèque de classes Java (JCL) pour créer des miniatures à partir de fichiers d'image chargés.
- layer-java : fonction Java qui illustre comment utiliser une couche Lambda pour empaqueter les dépendances séparément du code de votre fonction principale.

Tous les exemples d'applications ont un suivi actif activé pour les fonctions Lambda. Par exemple, l'`s3-java`application montre l'instrumentation automatique des AWS SDK for Java 2.x clients, la gestion des segments pour les tests, les sous-segments personnalisés et l'utilisation de couches Lambda pour stocker les dépendances d'exécution.

Exemples d'applications Java pour AWS Lambda

Le GitHub référentiel de ce guide fournit des exemples d'applications illustrant l'utilisation de Java dans AWS Lambda. Chaque exemple d'application inclut des scripts pour faciliter le déploiement et le nettoyage, un CloudFormation modèle et des ressources de support.

Exemples d’applications Lambda en Java

- example-java : fonction Java qui montre comment utiliser Lambda pour traiter les commandes. Cette fonction montre comment définir et désérialiser un objet d'événement d'entrée personnalisé, utiliser le AWS SDK et enregistrer les sorties.
- java-basic – Ensemble de fonctions Java minimales avec des tests unitaires et une configuration de journalisation variable.
- java events – Ensemble de fonctions Java contenant du code squelette permettant de gérer les événements de divers services tels qu’Amazon API Gateway, Amazon SQS et Amazon Kinesis. Ces fonctions utilisent la dernière version de la aws-lambda-java-eventsbibliothèque (3.0.0 et versions ultérieures). Ces exemples ne nécessitent pas le AWS SDK en tant que dépendance.
- s3-java – Fonction Java qui traite les événements de notification d’Amazon S3 et utilise la bibliothèque de classes Java (JCL) pour créer des miniatures à partir de fichiers d’image chargés.
- layer-java : fonction Java qui illustre comment utiliser une couche Lambda pour empaqueter les dépendances séparément du code de votre fonction principale.

Exécution de cadres Java populaires sur Lambda

- spring-cloud-function-samples— Un exemple tiré de Spring qui montre comment utiliser le framework Spring Cloud Function pour créer des fonctions AWS Lambda.
- Démo de l'application Spring Boot sans serveur : exemple qui montre comment configurer une application Spring Boot typique dans un environnement d'exécution Java géré avec ou sans SnapStart, ou en tant qu'image native de GraalVM avec un environnement d'exécution personnalisé.
- Démonstration de l'application Micronaut sans serveur — Un exemple qui montre comment utiliser Micronaut dans un environnement d'exécution Java géré avec et sans SnapStart, ou en tant qu'image native de GraalVM avec un environnement d'exécution personnalisé. Pour en savoir plus, consultez les guides Micronaut/Lambda.
- Démo de l'application Quarkus sans serveur — Un exemple qui montre comment utiliser Quarkus dans un environnement d'exécution Java géré avec et sans SnapStart, ou en tant qu'image native

de GraalVM avec un environnement d'exécution personnalisé. Pour en savoir plus, consultez le guide Quarkus/Lambda et le guide Quarkus/Lambda. SnapStart

Si vous débutez avec les fonctions Lambda dans Java, commencez par les exemples `java-basic`. Pour démarrer avec les sources d'événement Lambda, consultez les exemples `java-events`. Ces deux ensembles d'exemples montrent l'utilisation des bibliothèques Java, des variables d'environnement, du SDK et du AWS SDK de Lambda. AWS X-Ray Ces exemples nécessitent une configuration minimale et peuvent être déployé à partir de la ligne de commande en moins d'une minute.

Création de fonctions Lambda avec Go

Go est implémenté différemment des autres exécutions gérées. Go se compilant nativement en un binaire exécutable, il n'a pas besoin d'un environnement d'exécution dédié au langage. Utilisez un environnement d'exécution réservé au système d'exploitation (la famille d'environnement d'exécution `provided`) pour déployer les fonctions Go sur Lambda.

Rubriques

- Prise en charge de l'exécution Go
- Outils et bibliothèques
- Définition du gestionnaire de fonction Lambda dans Go
- Utilisation de l'objet de contexte Lambda pour récupérer les informations de la fonction Go
- Déployer des fonctions Lambda Go avec des archives de fichiers .zip
- Déployer des fonctions Lambda Go avec des images conteneurs
- Utilisation de couches pour les fonctions Lambda Go
- Journalisation et surveillance des fonctions Lambda Go
- Instrumentation du code Go dans AWS Lambda

Prise en charge de l'exécution Go

L'environnement d'exécution géré par Go 1.x pour Lambda est obsolète. Si vous utilisez l'environnement d'exécution Go 1.x, vous devez transférer vos fonctions vers `provided.al2023` ou `provided.al2`. Les environnements d'`provided.al2`exécution `provided.al2023` et offrent plusieurs avantages`go1.x`, notamment la prise en charge de l'architecture arm64 (processeurs AWS Graviton2), des fichiers binaires plus petits et des temps d'appel légèrement plus rapides.

Aucune modification du code n'est requise pour cette migration. Les seules modifications requises concernent la façon dont vous créez votre package de déploiement et l'exécution que vous utilisez pour créer votre fonction. Pour plus d'informations, consultez la section Migration des AWS Lambda fonctions de l'environnement d'exécution Go1.x vers le runtime personnalisé sur Amazon Linux 2 sur le AWS blog Compute.

Name	Identifiant	Système d'exploitation	Date d'obsolescence	Créer la fonction de blocage	Mettre à jour la fonction de blocage
Exécution réservée au système d'exploitation	`provided.al2023`	Amazon Linux 2	30 juin 2029	31 juillet 2029	31 août 2029
Exécution réservée au système d'exploitation	`provided.al2`	Amazon Linux 2	31 juillet 2026	31 août 2026	30 sept. 2026

Outils et bibliothèques

Lambda fournit les outils et bibliothèques suivants pour l'environnement d'exécution Go :

- AWS SDK pour Go v2 : Le AWS SDK officiel pour le langage de programmation Go.
- github. com/aws/aws-lambda-go/lambda: L'implémentation du modèle de programmation Lambda pour Go. Ce package est utilisé AWS Lambda pour appeler votre gestionnaire.
- github. com/aws/aws-lambda-go/lambdacontext: aides pour accéder aux informations contextuelles à partir de l'objet de contexte.
- github. com/aws/aws-lambda-go/events: Cette bibliothèque fournit des définitions de type pour les intégrations de sources d'événements courantes.
- github. com/aws/aws-lambda-go/cmd/build-lambda-zip : Cet outil peut être utilisé pour créer une archive de fichiers .zip sous Windows.

Pour plus d'informations, voir aws-lambda-goci-dessous GitHub.

Lambda fournit les exemples d'applications suivants pour l'environnement d'exécution Go :

Exemples d'applications Lambda en Go

- go-al2 – Une fonction Hello World qui renvoie l'adresse IP publique. Cette application utilise l'exécution personnalisée `provided.al2`.

- blank-go — Une fonction Go qui montre l'utilisation des bibliothèques Go de Lambda, de la journalisation, des variables d'environnement et du SDK. AWS Cette application utilise l’exécution `go1.x`.

Définition du gestionnaire de fonction Lambda dans Go

Le gestionnaire de fonction Lambda est la méthode dans votre code de fonction qui traite les événements. Lorsque votre fonction est invoquée, Lambda exécute la méthode du gestionnaire. Votre fonction s'exécute jusqu'à ce que le gestionnaire renvoie une réponse, se ferme ou expire.

Cette page explique comment utiliser les gestionnaires de fonctions Lambda dans Go, notamment la configuration du projet, les conventions de dénomination et les pratiques exemplaires. Cette page inclut également un exemple de fonction Lambda Go qui collecte des informations relatives à une commande, produit un reçu sous forme de fichier texte et place ce fichier dans un compartiment Amazon Simple Storage Service (Amazon S3). Pour plus d'informations sur le déploiement de votre fonction après l'avoir écrite, consultez the section called "Déploiement d'archives de fichiers .zip" ou the section called "Déploiement d'images de conteneur".

Rubriques

- Configuration de votre projet de gestionnaire Go
- Exemple de fonction Lambda Go
- Convention de nommage du gestionnaire
- Définition et accès à l'objet d'événement d'entrée
- Accès et utilisation de l'objet de contexte Lambda
- Signatures de gestionnaire valides pour les gestionnaires Go
- Utilisation de l'AWS SDK pour Go v2 dans votre gestionnaire
- Accès aux variables d'environnement
- Utilisation de l'état global
- Pratiques exemplaires en matière de code pour les fonctions Lambda Go

Configuration de votre projet de gestionnaire Go

Une fonction Lambda écrite en Go est créée en tant qu'exécutable Go. Vous pouvez initialiser un projet de fonction Lambda Go de la même manière que vous initialisez n'importe quel autre projet Go à l'aide de la commande suivante : `go mod init`

```
go mod init example-go
```

Ici, `example-go` est le nom du module. Vous pouvez le remplacer par n'importe quelle valeur. Cette commande initialise votre projet et génère le fichier `go.mod` répertoriant les dépendances de votre projet.

Utilisez la commande `go get` pour ajouter des dépendances externes à votre projet. Par exemple, pour toutes les fonctions Lambda en Go, vous devez inclure le package github.com/aws/aws-lambda-go/lambda, qui met en œuvre le modèle de programmation Lambda pour Go. Installez ce package à l'aide de la commande `go get` suivante :

```
go get github.com/aws/aws-lambda-go
```

Le code de votre fonction doit résider dans un fichier Go. Dans l'exemple suivant, nous nommons ce fichier `main.go`. Dans ce fichier, vous implémentez la logique de votre fonction de base dans une méthode de gestion, ainsi que dans une fonction `main()` qui appelle ce gestionnaire.

Exemple de fonction Lambda Go

L'exemple de code de fonction Lambda Go suivant prend en compte les informations relatives à une commande, produit un reçu sous forme de fichier texte et place ce fichier dans un compartiment Amazon S3.

Example Fonction Lambda **`main.go`**

```
package main

import (
 "context"
 "encoding/json"
 "fmt"
 "log"
 "os"
 "strings"

 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/s3"
)

type Order struct {
 OrderID string  `json:"order_id"`
 Amount  float64 `json:"amount"`
```

```
 Item    string  `json:"item"`
}

var (
 s3Client *s3.Client
)

func init() {
 // Initialize the S3 client outside of the handler, during the init phase
 cfg, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  log.Fatalf("unable to load SDK config, %v", err)
 }

 s3Client = s3.NewFromConfig(cfg)
}

func uploadReceiptToS3(ctx context.Context, bucketName, key, receiptContent string)
 error {
 _, err := s3Client.PutObject(ctx, &s3.PutObjectInput{
  Bucket: &bucketName,
  Key:    &key,
  Body:   strings.NewReader(receiptContent),
 })
 if err != nil {
  log.Printf("Failed to upload receipt to S3: %v", err)
  return err
 }
 return nil
}

func handleRequest(ctx context.Context, event json.RawMessage) error {
 // Parse the input event
 var order Order
 if err := json.Unmarshal(event, &order); err != nil {
  log.Printf("Failed to unmarshal event: %v", err)
  return err
 }

 // Access environment variables
 bucketName := os.Getenv("RECEIPT_BUCKET")
 if bucketName == "" {
  log.Printf("RECEIPT_BUCKET environment variable is not set")
  return fmt.Errorf("missing required environment variable RECEIPT_BUCKET")
```

```
 }

 // Create the receipt content and key destination
 receiptContent := fmt.Sprintf("OrderID: %s\nAmount: $%.2f\nItem: %s",
  order.OrderID, order.Amount, order.Item)
 key := "receipts/" + order.OrderID + ".txt"

 // Upload the receipt to S3 using the helper method
 if err := uploadReceiptToS3(ctx, bucketName, key, receiptContent); err != nil {
  return err
 }

 log.Printf("Successfully processed order %s and stored receipt in S3 bucket %s",
 order.OrderID, bucketName)
 return nil
}

func main() {
 lambda.Start(handleRequest)
}
```

Ce fichier `main.go` comprend les sections suivantes :

- `package main` : dans Go, le package contenant votre fonction `func main()` doit toujours être nommé `main`.
- Bloc `import` : utilisez ce bloc pour inclure les bibliothèques que votre fonction Lambda requiert.
- Bloc `type Order struct {}` : définissez la forme de l'événement d'entrée attendu dans cette structure Go.
- Bloc `var ()` : utilisez ce bloc pour définir les variables globales que vous utiliserez dans votre fonction Lambda.
- `func init() {}` : incluez tout code que vous voulez que Lambda exécute pendant la phase d'initialisation dans cette méthode `init()`.
- `func uploadReceiptToS3(...) {}` : il s'agit d'une méthode auxiliaire référencée par la méthode de gestion principale `handleRequest`.
- `func handleRequest(ctx context.Context, event json.RawMessage) error {}` : il s'agit de la méthode de gestion principale, qui contient la logique principale de votre application.
- `func main() {}` : il s'agit d'un point d'entrée obligatoire pour votre gestionnaire Lambda. L'argument de la méthode `lambda.Start()` est votre méthode de gestion principale.

Pour que cette fonction fonctionne correctement, son rôle d'exécution doit autoriser l'action `s3:PutObject`. Assurez-vous également de définir la variable d'environnement `RECEIPT_BUCKET`. Après une invocation réussie, le compartiment Amazon S3 doit contenir un fichier de reçu.

Convention de nommage du gestionnaire

Pour les fonctions Lambda dans Go, vous pouvez utiliser n'importe quel nom pour le gestionnaire. Dans cet exemple, le nom de la méthode de traitement est `handleRequest`. Pour référencer la valeur du gestionnaire dans votre code, vous pouvez utiliser la variable d'environnement `_HANDLER`.

Pour les fonctions Go qui utilisent l'exécution dans un package de déploiement .zip, le fichier exécutable qui contient le code de votre fonction doit être nommé `bootstrap`. De plus, le fichier `bootstrap` doit se trouver à la racine du fichier .zip. Pour les fonctions Go déployées à l'aide d'une image de conteneur, vous pouvez utiliser n'importe quel nom pour le fichier exécutable.

Définition et accès à l'objet d'événement d'entrée

JSON est le format d'entrée le plus courant et standard pour les fonctions Lambda. Dans cet exemple, la fonction exige une entrée similaire à l'exemple suivant :

```
{
    "order_id": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

Lorsque vous utilisez des fonctions Lambda dans Go, vous pouvez définir la forme de l'événement d'entrée attendu sous la forme d'une structure Go. Dans cet exemple, nous définissons une structure pour représenter un `Order` :

```
type Order struct {
    OrderID string  `json:"order_id"`
    Amount  float64 `json:"amount"`
    Item    string  `json:"item"`
}
```

Cette structure correspond à la forme d'entrée attendue. Après avoir défini votre structure, vous pouvez écrire une signature de gestionnaire qui utilise un type JSON générique compatible avec la bibliothèque standard encoding/json. Vous pouvez ensuite le désérialiser dans votre structure à l'aide de la fonction func Unmarshal. Ceci est illustré dans les premières lignes du gestionnaire :

```
func handleRequest(ctx context.Context, event json.RawMessage) error {
    // Parse the input event
    var order Order
    if err := json.Unmarshal(event, &order); err != nil {
        log.Printf("Failed to unmarshal event: %v", err)
        return err
    ...
}
```

Après cette désérialisation, vous pouvez accéder aux champs de la variable `order`. Par exemple, `order.OrderID` récupère la valeur de `"order_id"` à partir de l'entrée d'origine.

> **Note**
>
> Le package `encoding/json` ne peut accéder qu'aux champs exportés. Pour être exportées, les noms de champs de la structure d'événement doivent être en majuscules.

Accès et utilisation de l'objet de contexte Lambda

L'objet de contexte Lambda contient des informations sur l'invocation, la fonction et l'environnement d'exécution. Dans cet exemple, nous avons déclaré cette variable comme `ctx` dans la signature du gestionnaire :

```
func handleRequest(ctx context.Context, event json.RawMessage) error {
    ...
}
```

L'entrée `ctx context.Context` est un argument facultatif dans votre gestionnaire de fonctions. Pour plus d'informations sur les signatures de gestionnaire acceptées, consultez the section called "Signatures de gestionnaire valides pour les gestionnaires Go".

Si vous appelez d'autres services à l'aide du kit SDK AWS, l'objet de contexte est requis dans quelques domaines clés. Par exemple, pour initialiser correctement vos clients SDK, vous pouvez charger la configuration du kit SDK AWS appropriée à l'aide de l'objet de contexte comme suit :

```
// Load AWS SDK configuration using the default credential provider chain
    cfg, err := config.LoadDefaultConfig(ctx)
```

Les appels du kit SDK eux-mêmes peuvent nécessiter l'objet de contexte comme entrée. Par exemple, l'appel `s3Client.PutObject` accepte l'objet de contexte comme premier argument :

```
// Upload the receipt to S3
    _, err = s3Client.PutObject(ctx, &s3.PutObjectInput{
        ...
    })
```

En dehors des requêtes du kit SDK AWS, vous pouvez également utiliser l'objet de contexte pour surveiller les fonctions. Pour plus d'informations sur la copie d'objets, consultez the section called "Contexte".

Signatures de gestionnaire valides pour les gestionnaires Go

Vous disposez de plusieurs options lorsque vous créez un gestionnaire de fonction Lambda dans Go, mais vous devez respecter les règles suivantes :

- Le gestionnaire doit être une fonction.
- Le gestionnaire peut accepter entre 0 et 2 arguments. S'il y a deux arguments, le premier argument doit implémenter `context.Context`.
- Le gestionnaire peut retourner entre 0 et 2 arguments. S'il y a une seule valeur renvoyée, elle doit implémenter `error`. S'il y a deux valeurs renvoyées, la seconde valeur doit implémenter `error`.

Voici la liste des signatures de gestionnaire valides. `TIn` et `TOut` représentent des types compatibles avec la bibliothèque encoding/json standard. Pour en savoir plus, reportez-vous à la section func Unmarshal pour savoir comment ces types sont désérialisés.

- `func ()`
- `func () error`
- `func () (TOut, error)`
- `func (TIn) error`
- `func (TIn) (TOut, error)`

- `func (context.Context) error`
- `func (context.Context) (TOut, error)`
- `func (context.Context, TIn) error`
- `func (context.Context, TIn) (TOut, error)`

Utilisation de l'AWS SDK pour Go v2 dans votre gestionnaire

Vous utiliserez souvent les fonctions Lambda pour interagir avec d'autres ressources AWS ou pour les mettre à jour. Le moyen le plus simple d'interagir avec ces ressources est d'utiliser l'AWS SDK pour Go v2.

> **Note**
>
> L'AWS SDK pour Go (v1) est en mode de maintenance et atteindra la fin du support le 31 juillet 2025. Nous vous recommandons de n'utiliser que l'AWS SDK pour Go v2 à l'avenir.

Pour ajouter des dépendances au SDK à votre fonction, utilisez la commande `go get` correspondant aux clients SDK spécifiques dont vous avez besoin. Dans l'exemple de code précédent, nous avons utilisé les bibliothèques `config` et `s3`. Ajoutez ces dépendances en exécutant les commandes suivantes dans le répertoire qui contient vos fichiers `go.mod` et `main.go` :

```
go get github.com/aws/aws-sdk-go-v2/config
go get github.com/aws/aws-sdk-go-v2/service/s3
```

Importez ensuite les dépendances en conséquence dans le bloc d'importation de votre fonction :

```
import (
    ...
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/s3"
)
```

Lorsque vous utilisez le kit SDK dans votre gestionnaire, configurez vos clients avec les bons paramètres. La méthode la plus simple consiste à utiliser la chaîne de fournisseurs d'informations d'identification par défaut. Cet exemple illustre une manière de charger cette configuration :

```
// Load AWS SDK configuration using the default credential provider chain
    cfg, err := config.LoadDefaultConfig(ctx)
    if err != nil {
        log.Printf("Failed to load AWS SDK config: %v", err)
        return err
    }
```

Après avoir chargé cette configuration dans la variable `cfg`, vous pouvez transmettre cette variable aux instanciations du client. L'exemple de code instancie un client Amazon S3 comme suit :

```
// Create an S3 client
    s3Client := s3.NewFromConfig(cfg)
```

Dans cet exemple, nous avons initialisé notre client Amazon S3 dans la fonction `init()` pour éviter d'avoir à l'initialiser à chaque fois que nous invoquons notre fonction. Le problème est que dans la fonction `init()`, Lambda n'a pas accès à l'objet de contexte. Pour contourner le problème, vous pouvez transmettre un espace réservé, comme `context.TODO()`, lors de la phase d'initialisation. Plus tard, lorsque vous passez un appel à l'aide du client, transmettez l'objet de contexte complet. Cette solution est également décrite dans the section called "Utilisation du contexte dans les initialisations et les appels du client du kit SDK AWS".

Après avoir configuré et initialisé votre client SDK, vous pouvez l'utiliser pour interagir avec d'autres services AWS. L'exemple de code appelle l'API Amazon S3 `PutObject` comme suit :

```
_, err = s3Client.PutObject(ctx, &s3.PutObjectInput{
    Bucket: &bucketName,
    Key:    &key,
    Body:   strings.NewReader(receiptContent),
})
```

Accès aux variables d'environnement

Dans le code de votre gestionnaire, vous pouvez référencer n'importe quelle variable d'environnement à l'aide de la méthode `os.Getenv()`. Dans cet exemple, nous référençons la variable d'environnement `RECEIPT_BUCKET` définie à l'aide de la ligne de code suivante :

```
// Access environment variables
    bucketName := os.Getenv("RECEIPT_BUCKET")
    if bucketName == "" {
        log.Printf("RECEIPT_BUCKET environment variable is not set")
        return fmt.Errorf("missing required environment variable RECEIPT_BUCKET")
    }
```

Utilisation de l'état global

Pour éviter de créer des ressources à chaque fois que vous invoquez votre fonction, vous pouvez déclarer et modifier les variables globales en dehors du code de votre gestionnaire de fonction Lambda. Vous définissez ces variables globales dans un bloc ou une instruction `var`. De plus, votre gestionnaire peut déclarer une fonction `init()` qui est exécutée pendant la phase d'initialisation. La méthode `init` se comporte de la même manière dans AWS Lambda que dans les programmes Go standard.

Pratiques exemplaires en matière de code pour les fonctions Lambda Go

Respectez les directives de la liste suivante pour utiliser les pratiques exemplaires de codage lors de la création de vos fonctions Lambda :

- Séparez le gestionnaire Lambda de votre logique principale. Cela vous permet de créer une fonction testable plus unitaire.
- Réduisez la complexité de vos dépendances. Privilégiez les infrastructures plus simples qui se chargent rapidement au démarrage de l'environnement d'exécution.
- Réduisez la taille de votre package de déploiement selon ses besoins relatifs à l'environnement d'exécution. Cela contribue à réduire le temps nécessaire au téléchargement et à la décompression de votre package de déploiement avant l'invocation.

Tirez parti de la réutilisation de l'environnement d'exécution pour améliorer les performances de votre fonction. Initialisez les clients SDK et les connexions à la base de données en dehors du gestionnaire de fonctions et mettez en cache les actifs statiques localement dans le répertoire `/tmp`. Les invocations ultérieures traitées par la même instance de votre fonction peuvent réutiliser ces ressources. Cela permet d'économiser des coûts, tout en réduisant le temps d'exécution de la fonction.

Pour éviter des éventuelles fuites de données entre les invocations, n'utilisez pas l'environnement d'exécution pour stocker des données utilisateur, des événements ou d'autres informations ayant un

impact sur la sécurité. Si votre fonction repose sur un état réversible qui ne peut pas être stocké en mémoire dans le gestionnaire, envisagez de créer une fonction distincte ou des versions distinctes d'une fonction pour chaque utilisateur.

Utilisez une directive keep-alive pour maintenir les connexions persistantes. Lambda purge les connexions inactives au fil du temps. Si vous tentez de réutiliser une connexion inactive lorsque vous invoquez une fonction, cela entraîne une erreur de connexion. Pour maintenir votre connexion persistante, utilisez la directive Keep-alive associée à votre environnement d'exécution. Pour obtenir un exemple, consultez Réutilisation des connexions avec Keep-Alive dans Node.js.

Utilisez des variables d'environnement pour transmettre des paramètres opérationnels à votre fonction. Par exemple, si vous écrivez dans un compartiment Amazon S3 au lieu de coder en dur le nom du compartiment dans lequel vous écrivez, configurez le nom du compartiment comme variable d'environnement.

Évitez d'utiliser des invocations récursives dans votre fonction Lambda, lorsque la fonction s'invoque elle-même ou démarre un processus susceptible de l'invoquer à nouveau. Cela peut entraîner un volume involontaire d'invocations de fonction et des coûts accrus. Si vous constatez un volume involontaire d'invocations, définissez immédiatement la simultanéité réservée à la fonction sur `0` afin de limiter toutes les invocations de la fonction, pendant que vous mettez à jour le code.

N'utilisez pas d'API non publiques non documentées dans le code de votre fonction Lambda. Pour les exécutions gérées AWS Lambda, Lambda applique périodiquement des mises à jour de sécurité et fonctionnelles aux API internes de Lambda. Ces mises à jour internes de l'API peuvent être incompatibles avec les versions antérieures, entraînant des conséquences imprévues telles que des échecs d'invocation si votre fonction dépend de ces API non publiques. Consultez la Référence d'API pour obtenir la liste des API accessibles au public.

Écriture du code idempotent. L'écriture de code idempotent pour vos fonctions garantit ne gestion identique des événements dupliqués. Votre code doit valider correctement les événements et gérer correctement les événements dupliqués. Pour de plus amples informations, veuillez consulterComment faire en sorte que ma fonction Lambda soit idempotente ?.

Utilisation de l'objet de contexte Lambda pour récupérer les informations de la fonction Go

Dans Lambda, l'objet de contexte fournit aux méthodes et aux propriétés des informations sur l'invocation, la fonction et l'environnement d'exécution. Lorsque Lambda exécute votre fonction, il transmet un objet contexte au gestionnaire. Pour utiliser l'objet de contexte dans votre gestionnaire, vous pouvez éventuellement le déclarer comme paramètre d'entrée dans votre gestionnaire. L'objet de contexte est nécessaire si vous souhaitez effectuer les opérations suivantes dans votre gestionnaire :

- Vous devez accéder à toutes les variables, méthodes ou propriétés globales proposées par l'objet de contexte. Ces méthodes et propriétés sont utiles pour des tâches telles que la détermination de l'entité qui a appelé votre fonction ou la mesure du temps d'invocation de votre fonction, comme illustré dans the section called "Accès aux informations du contexte d'appel".
- Vous devez utiliser l'AWS SDK pour Go pour effectuer des appels à d'autres services. L'objet de contexte est un paramètre d'entrée important pour la plupart de ces appels. Pour de plus amples informations, consultez the section called "Utilisation du contexte dans les initialisations et les appels du client du kit SDK AWS".

Rubriques

- Variables, méthodes et propriétés prises en charge par l'objet de contexte
- Accès aux informations du contexte d'appel
- Utilisation du contexte dans les initialisations et les appels du client du kit SDK AWS

Variables, méthodes et propriétés prises en charge par l'objet de contexte

La bibliothèque de contexte Lambda fournit les variables globales, méthodes et propriétés suivantes.

Variables globales

- `FunctionName` – Nom de la fonction Lambda.
- `FunctionVersion` – Version de la fonction.
- `MemoryLimitInMB` – Quantité de mémoire allouée à la fonction.
- `LogGroupName` – Groupe de journaux pour la fonction.

- `LogStreamName` – Flux de journal de l'instance de fonction.

Méthodes de contexte

- `Deadline` – Retourne la date d'expiration de l'exécution, exprimée en millisecondes au format horaire Unix.

Propriétés du contexte

- `InvokedFunctionArn` – Amazon Resource Name (ARN) utilisé pour appeler la fonction. Indique si l'appelant a spécifié un numéro de version ou un alias.
- `AwsRequestID` – Identifiant de la demande d'appel.
- `Identity` – (applications mobiles) Informations sur l'identité Amazon Cognito qui a autorisé la demande.
- `ClientContext` – (applications mobiles) Contexte client fourni à Lambda par l'application client.

Accès aux informations du contexte d'appel

Les fonctions Lambda ont accès aux métadonnées sur leur environnement et la demande d'appel. Elles sont accessibles à l'adresse du contexte du package. Si votre gestionnaire inclut `context.Context` en tant que paramètre, Lambda insère les informations sur votre fonction dans la propriété `Value` du contexte. Notez que vous devez importer la bibliothèque `lambdacontext` pour accéder au contenu de l'objet `context.Context`.

```
package main

import (
        "context"
        "log"
        "github.com/aws/aws-lambda-go/lambda"
        "github.com/aws/aws-lambda-go/lambdacontext"
)

func CognitoHandler(ctx context.Context) {
        lc, _ := lambdacontext.FromContext(ctx)
        log.Print(lc.Identity.CognitoIdentityPoolID)
}
```

```
func main() {
        lambda.Start(CognitoHandler)
}
```

Dans l'exemple ci-dessus, `lc` est la variable utilisée pour consommer les informations que l'objet de contexte a capturées et `log.Print(lc.Identity.CognitoIdentityPoolID)` affiche ces informations, dans ce cas, le CognitoPoolID.

L'exemple suivant présente la façon d'utiliser l'objet contexte pour surveiller le temps nécessaire à l'exécution de votre fonction Lambda. Cela vous permet d'analyser les attentes de performance et d'ajuster le code de votre fonction en conséquence, si nécessaire.

```
package main

import (
        "context"
        "log"
        "time"
        "github.com/aws/aws-lambda-go/lambda"
)

func LongRunningHandler(ctx context.Context) (string, error) {

        deadline, _ := ctx.Deadline()
        deadline = deadline.Add(-100 * time.Millisecond)
        timeoutChannel := time.After(time.Until(deadline))

        for {

                select {

                case <- timeoutChannel:
                        return "Finished before timing out.", nil

                default:
                        log.Print("hello!")
                        time.Sleep(50 * time.Millisecond)
                }
        }
}

func main() {
        lambda.Start(LongRunningHandler)
```

```
}
```

Utilisation du contexte dans les initialisations et les appels du client du kit SDK AWS

Si votre gestionnaire doit utiliser l'AWS SDK pour Go pour appeler d'autres services, incluez l'objet de contexte en tant qu'entrée dans votre gestionnaire. Dans AWS, il est recommandé de transmettre l'objet de contexte dans la plupart des appels au kit SDK AWS. Par exemple, l'appel Amazon S3 `PutObject` accepte l'objet de contexte (`ctx`) comme premier argument :

```
// Upload an object to S3
    _, err = s3Client.PutObject(ctx, &s3.PutObjectInput{
        ...
    })
```

Pour initialiser correctement vos clients SDK, vous pouvez également utiliser l'objet de contexte pour charger la configuration correcte avant de transmettre cet objet de configuration au client :

```
// Load AWS SDK configuration using the default credential provider chain
    cfg, err := config.LoadDefaultConfig(ctx)
    ...
    s3Client = s3.NewFromConfig(cfg)
```

Si vous souhaitez initialiser vos clients SDK en dehors de votre gestionnaire principal (c'est-à-dire pendant la phase d'initialisation), vous pouvez transmettre un objet de contexte de remplacement :

```
func init() {
 // Initialize the S3 client outside of the handler, during the init phase
 cfg, err := config.LoadDefaultConfig(context.TODO())
 ...
 s3Client = s3.NewFromConfig(cfg)
}
```

Si vous initialisez vos clients de cette façon, assurez-vous de transmettre le bon objet de contexte dans les appels du SDK depuis votre gestionnaire principal.

Déployer des fonctions Lambda Go avec des archives de fichiers .zip

Le code de votre fonction AWS Lambda se compose de scripts ou de programmes compilés et leurs dépendances. Pour déployer votre code de fonction vers Lambda, vous utilisez un package de déploiement. Lambda prend en charge deux types de packages de déploiement : les images conteneurs et les archives de fichiers .zip.

Cette page explique comment créer un fichier .zip en tant que package de déploiement pour l'exécution Go, puis utiliser le fichier .zip pour déployer votre code de fonction sur AWS Lambda à l'aide de la AWS Management Console, AWS Command Line Interface (AWS CLI) et de AWS Serverless Application Model (AWS SAM).

Notez que Lambda utilise les autorisations de fichiers POSIX. Ainsi, vous pourriez devoir définir des autorisations pour le dossier du package de déploiement avant de créer l'archive de fichiers .zip.

Sections

- Création d'un fichier .zip sur macOS et Linux
- Création d'un fichier .zip sous Windows
- Création et mise à jour de fonctions Lambda Go à l'aide de fichiers .zip

Création d'un fichier .zip sur macOS et Linux

Les étapes suivantes montrent comment compiler votre exécutable à l'aide de la commande go `build` et créer un package de déploiement de fichiers .zip pour Lambda. Avant de compiler votre code, assurez-vous d'avoir installé le package lambda depuis GitHub. Ce module fournit une implémentation de l'interface d'exécution, qui gère l'interaction entre Lambda et le code de votre fonction. Pour télécharger cette bibliothèque, exécutez la commande suivante.

```
go get github.com/aws/aws-lambda-go/lambda
```

Si votre fonction utilise le AWS SDK pour Go, téléchargez l'ensemble standard de modules de kit SDK, ainsi que tous les clients API de services AWS requis par votre application. Pour savoir comment installer le kit SDK pour Go, consultez Démarrer avec le AWS SDK pour Go V2.

Utilisation de la famille d'environnement d'exécution fournie

Go est implémenté différemment des autres exécutions gérées. Go se compilant nativement en un binaire exécutable, il n'a pas besoin d'un environnement d'exécution dédié au langage. Utilisez un environnement d'exécution réservé au système d'exploitation (la famille d'environnement d'exécution `provided`) pour déployer les fonctions Go sur Lambda.

Pour créer un package de déploiement .zip (macOS/Linux)

1. Dans le répertoire du projet qui contient le fichier `main.go` de votre application, compilez votre exécutable. Remarques :

   - L'exécutable doit être nommé `bootstrap`. Pour de plus amples informations, consultez Convention de nommage du gestionnaire.
   - Définissez l'architecture de jeu d'instructions cible. Les environnements d'exécution réservés au système d'exploitation prennent en charge les architectures arm64 et x86_64.
   - Vous pouvez utiliser la balise optionnelle `lambda.norpc` pour exclure le composant RPC (Remote Procedure Call) de la bibliothèque lambda. Le composant RPC n'est nécessaire que lorsque vous utilisez l'environnement d'exécution Go 1.x. L'exclusion du RPC réduit la taille du package de déploiement.

   Pour l'architecture arm64 :

   ```
   GOOS=linux GOARCH=arm64 go build -tags lambda.norpc -o bootstrap main.go
   ```

   Pour l'architecture x86_64 :

   ```
   GOOS=linux GOARCH=amd64 go build -tags lambda.norpc -o bootstrap main.go
   ```

2. (Facultatif) Vous devrez peut-être compiler des packages avec `CGO_ENABLED=0` configuré sur Linux :

   ```
   GOOS=linux GOARCH=arm64 CGO_ENABLED=0 go build -o bootstrap -tags lambda.norpc
    main.go
   ```

   Cette commande crée un package binaire stable pour les versions de bibliothèque C standard (`libc`), qui peuvent être différentes sur Lambda et d'autres appareils.
3. Créez un package de déploiement en empaquetant l'exécutable dans un fichier .zip.

```
zip myFunction.zip bootstrap
```

> **Note**
>
> Le fichier `bootstrap` doit se situer à la racine du fichier .zip.

4. Créez la fonction . Remarques :

   - Le binaire doit être nommé `bootstrap`, mais le nom du gestionnaire peut être quelconque. Pour de plus amples informations, consultez Convention de nommage du gestionnaire.
   - L'option `--architectures` n'est nécessaire que si vous utilisez arm64. La valeur par défaut est x86_64.
   - Pour `--role`, spécifiez l'Amazon Resource Name (ARN) du rôle d'exécution.

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--architectures arm64 \
--role arn:aws:iam::111122223333:role/lambda-ex \
--zip-file fileb://myFunction.zip
```

Création d'un fichier .zip sous Windows

Les étapes suivantes montrent comment télécharger l'outil build-lambda-zip pour Windows à partir de GitHub, compiler votre exécutable et créer un package de déploiement .zip.

> **Note**
>
> Si vous ne l'avez pas déjà fait, vous devez installer git, puis ajouter l'exécutable `git` à votre variable d'environnement Windows `%PATH%`.

Avant de compiler votre code, assurez-vous d'avoir installé la bibliothèque lambda depuis GitHub. Pour télécharger cette bibliothèque, exécutez la commande suivante.

```
go get github.com/aws/aws-lambda-go/lambda
```

Si votre fonction utilise le AWS SDK pour Go, téléchargez l'ensemble standard de modules de kit SDK, ainsi que tous les clients API de services AWS requis par votre application. Pour savoir comment installer le kit SDK pour Go, consultez Démarrer avec le AWS SDK pour Go V2.

Utilisation de la famille d'environnement d'exécution fournie

Go est implémenté différemment des autres exécutions gérées. Go se compilant nativement en un binaire exécutable, il n'a pas besoin d'un environnement d'exécution dédié au langage. Utilisez un environnement d'exécution réservé au système d'exploitation (la famille d'environnement d'exécution `provided`) pour déployer les fonctions Go sur Lambda.

Pour créer un package de déploiement .zip (Windows)

1. Téléchargez l'outil build-lambda-zip à partir de GitHub.

   ```
   go install github.com/aws/aws-lambda-go/cmd/build-lambda-zip@latest
   ```

2. Utilisez l'outil de votre `GOPATH` pour créer un fichier .zip. Si vous disposez d'une installation de Go par défaut, l'outil est généralement dans `%USERPROFILE%\Go\bin`. Dans le cas contraire, accédez à l'endroit où vous avez installé l'environnement d'exécution Go et effectuez l'une des instructions suivantes :

   cmd.exe

   Dans cmd.exe, exécutez l'une des commandes suivantes, en fonction de l'architecture de jeu d'instructions cible. Les environnements d'exécution réservés au système d'exploitation prennent en charge les architectures arm64 et x86_64.

   Vous pouvez utiliser la balise optionnelle `lambda.norpc` pour exclure le composant RPC (Remote Procedure Call) de la bibliothèque lambda. Le composant RPC n'est nécessaire que lorsque vous utilisez l'environnement d'exécution Go 1.x. L'exclusion du RPC réduit la taille du package de déploiement.

   Example — Pour l'architecture x86_64

   ```
   set GOOS=linux
   set GOARCH=amd64
   set CGO_ENABLED=0
   go build -tags lambda.norpc -o bootstrap main.go
   %USERPROFILE%\Go\bin\build-lambda-zip.exe -o myFunction.zip bootstrap
   ```

Example — Pour l'architecture arm64

```
set GOOS=linux
set GOARCH=arm64
set CGO_ENABLED=0
go build -tags lambda.norpc -o bootstrap main.go
%USERPROFILE%\Go\bin\build-lambda-zip.exe -o myFunction.zip bootstrap
```

PowerShell

Dans PowerShell, exécutez l'une des commandes suivantes, en fonction de l'architecture de jeu d'instructions cible. Les environnements d'exécution réservés au système d'exploitation prennent en charge les architectures arm64 et x86_64.

Vous pouvez utiliser la balise optionnelle `lambda.norpc` pour exclure le composant RPC (Remote Procedure Call) de la bibliothèque lambda. Le composant RPC n'est nécessaire que lorsque vous utilisez l'environnement d'exécution Go 1.x. L'exclusion du RPC réduit la taille du package de déploiement.

Pour l'architecture x86_64 :

```
$env:GOOS = "linux"
$env:GOARCH = "amd64"
$env:CGO_ENABLED = "0"
go build -tags lambda.norpc -o bootstrap main.go
~\Go\Bin\build-lambda-zip.exe -o myFunction.zip bootstrap
```

Pour l'architecture arm64 :

```
$env:GOOS = "linux"
$env:GOARCH = "arm64"
$env:CGO_ENABLED = "0"
go build -tags lambda.norpc -o bootstrap main.go
~\Go\Bin\build-lambda-zip.exe -o myFunction.zip bootstrap
```

3. Créez la fonction . Remarques :

- Le binaire doit être nommé `bootstrap`, mais le nom du gestionnaire peut être quelconque. Pour de plus amples informations, consultez Convention de nommage du gestionnaire.

- L'option `--architectures` n'est nécessaire que si vous utilisez arm64. La valeur par défaut est x86_64.
- Pour `--role`, spécifiez l'Amazon Resource Name (ARN) du rôle d'exécution.

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--architectures arm64 \
--role arn:aws:iam::111122223333:role/lambda-ex \
--zip-file fileb://myFunction.zip
```

Création et mise à jour de fonctions Lambda Go à l'aide de fichiers .zip

Une fois que vous avez créé votre package de déploiement .zip, vous pouvez l'utiliser pour créer une nouvelle fonction Lambda ou mettre à jour une fonction Lambda existante. Vous pouvez déployer votre package .zip à l'aide de la console Lambda, de l'AWS Command Line Interface, et de l'API Lambda. Vous pouvez également créer et mettre à jour des fonctions Lambda à l'aide de l'AWS Serverless Application Model (AWS SAM) et de CloudFormation.

La taille maximale d'un package de déploiement .zip pour Lambda est de 250 Mo (décompressé). Notez que cette limite s'applique à la taille combinée de tous les fichiers que vous chargez, y compris les couches Lambda.

Le runtime Lambda a besoin d'une autorisation pour lire les fichiers de votre package de déploiement. Dans la notation octale des autorisations Linux, Lambda a besoin de 644 autorisations pour les fichiers non exécutables (rw-r--r--) et de 755 autorisations (rwxr-xr-x) pour les répertoires et les fichiers exécutables.

Sous Linux et macOS, utilisez la commande `chmod` pour modifier les autorisations de fichiers sur les fichiers et les répertoires de votre package de déploiement. Par exemple, pour octroyer à un fichier non exécutable les autorisations correctes, exécutez la commande suivante.

```
chmod 644 <filepath>
```

Pour modifier les autorisations relatives aux fichiers dans Windows, voir Définir, afficher, modifier ou supprimer des autorisations sur un objet dans la documentation Microsoft Windows.

> **Note**
>
> Si vous n'octroyez pas à Lambda les autorisations dont il a besoin pour accéder aux répertoires de votre package de déploiement, Lambda définit les autorisations de ces répertoires à 755 (rwxr-xr-x).

Création et mise à jour de fonctions avec des fichiers .zip à l'aide de la console

Pour créer une nouvelle fonction, vous devez d'abord créer la fonction dans la console, puis charger votre archive .zip. Pour mettre à jour une fonction existante, ouvrez la page de votre fonction, puis suivez la même procédure pour ajouter votre fichier .zip mis à jour.

Si votre fichier .zip fait moins de 50 Mo, vous pouvez créer ou mettre à jour une fonction en chargeant le fichier directement à partir de votre ordinateur local. Pour les fichiers .zip de plus de 50 Mo, vous devez d'abord charger votre package dans un compartiment Amazon S3. Pour savoir comment charger un fichier dans un compartiment Amazon S3 à l'aide de la AWS Management Console, veuillez consulter la rubrique Premiers pas avec Amazon S3. Pour charger des fichiers à l'aide de l'AWS CLI, veuillez consulter la rubrique Déplacer des objets dans le Guide de l'utilisateur AWS CLI.

> **Note**
>
> Vous ne pouvez pas convertir une fonction d'image de conteneur existante pour utiliser une archive .zip. Vous devez créer une nouvelle fonction.

Pour créer une nouvelle fonction (console)

1. Ouvrez la page Fonctions de la console Lambda et choisissez Créer une fonction.
2. Choisissez Créer à partir de zéro.
3. Sous Informations de base, procédez comme suit :
   a. Pour Nom de la fonction, saisissez le nom de la fonction.
   b. Dans Runtime (Exécution), choisissez `provided.al2023`.
4. (Facultatif) Sous Permissions (Autorisations), développez Change default execution role (Modifier le rôle d'exécution par défaut). Vous pouvez créer un rôle d'exécution ou en utiliser un existant.

5. Choisissez Créer une fonction. Lambda crée une fonction de base « Hello world » à l'aide de l'exécution de votre choix.

Pour charger une archive .zip à partir de votre ordinateur local (console)

1. Sur la page Fonctions de la console Lambda, choisissez la fonction pour laquelle vous souhaitez charger le fichier .zip.
2. Sélectionnez l'onglet Code.
3. Dans le volet Source du code, choisissez Charger à partir de.
4. Choisissez Fichier .zip.
5. Pour charger un fichier .zip, procédez comme suit :
   a. Sélectionnez Charger, puis choisissez votre fichier .zip dans le sélecteur de fichiers.
   b. Choisissez Ouvrir.
   c. Choisissez Enregistrer.

Pour charger une archive .zip depuis un compartiment Amazon S3 (console)

1. Sur la page Fonctions de la console Lambda, choisissez la fonction pour laquelle vous souhaitez charger un nouveau fichier .zip.
2. Sélectionnez l'onglet Code.
3. Dans le volet Source du code, choisissez Charger à partir de.
4. Choisissez l'emplacement Amazon S3.
5. Collez l'URL du lien Amazon S3 de votre fichier .zip et choisissez Enregistrer.

Création et mise à jour de fonctions avec des fichiers .zip à l'aide de l'AWS CLI

Vous pouvez utiliser l'AWS CLI pour créer une nouvelle fonction ou pour mettre à jour une fonction existante à l'aide d'un fichier .zip. Utilisez les commandes create-function et update-function-code pour déployer votre package .zip. Si votre fichier .zip est inférieur à 50 Mo, vous pouvez charger le package .zip à partir d'un emplacement de fichier sur votre machine de génération locale. Pour les fichiers plus volumineux, vous devez charger votre package .zip à partir d'un compartiment Amazon S3. Pour savoir comment charger un fichier dans un compartiment Amazon S3 à l'aide de l'AWS CLI, veuillez consulter la rubrique Déplacer des objets dans le Guide de l'utilisateur AWS CLI.

> **Note**
>
> Si vous chargez votre fichier .zip à partir d'un compartiment Amazon S3 à l'aide de l'AWS CLI, le compartiment doit se trouver dans la même Région AWS que votre fonction.

Pour créer une nouvelle fonction à l'aide d'un fichier .zip avec l'AWS CLI, vous devez spécifier les éléments suivants :

- Le nom de votre fonction (`--function-name`)
- L'exécution de votre fonction (`--runtime`)
- L'Amazon Resource Name (ARN) du rôle d'exécution de votre fonction (`--role`)
- Le nom de la méthode du gestionnaire dans votre code de fonction (`--handler`)

Vous devez également indiquer l'emplacement de votre fichier .zip. Si votre fichier .zip se trouve dans un dossier sur votre machine de génération locale, utilisez l'option `--zip-file` pour spécifier le chemin d'accès du fichier, comme le montre l'exemple de commande suivant.

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

Pour spécifier l'emplacement du fichier .zip dans un compartiment Amazon S3, utilisez l'option `--code` comme le montre l'exemple de commande suivant. Vous devez uniquement utiliser le paramètre `S3ObjectVersion` pour les objets soumis à la gestion des versions.

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

Pour mettre à jour une fonction existante à l'aide de l'interface de ligne de commande, vous devez spécifier le nom de votre fonction à l'aide du paramètre `--function-name`. Vous devez également spécifier l'emplacement du fichier .zip que vous souhaitez utiliser pour mettre à jour votre code de fonction. Si votre fichier .zip se trouve dans un dossier sur votre machine de génération locale,

utilisez l'option `--zip-file` pour spécifier le chemin d'accès du fichier, comme le montre l'exemple de commande suivant.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Pour spécifier l'emplacement du fichier .zip dans un compartiment Amazon S3, utilisez les options `--s3-bucket` et `--s3-key` comme le montre l'exemple de commande suivant. Vous devez uniquement utiliser le paramètre `--s3-object-version` pour les objets soumis à la gestion des versions.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Création et mise à jour de fonctions avec des fichiers .zip à l'aide de l'API Lambda

Pour créer et mettre à jour des fonctions à l'aide d'une archive de fichiers .zip, utilisez les opérations d'API suivantes :

- CreateFunction
- UpdateFunctionCode

Création et mise à jour de fonctions avec des fichiers .zip à l'aide d'AWS SAM

L'AWS Serverless Application Model (AWS SAM) est une boîte à outils qui permet de rationaliser le processus de création et d'exécution d'applications sans serveur sur AWS. Vous définissez les ressources de votre application dans un modèle YAML ou JSON et vous utilisez l'interface de ligne de commande AWS SAM (AWS SAM CLI) pour créer, empaqueter et déployer vos applications. Lorsque vous créez une fonction Lambda à partir d'un modèle AWS SAM, AWS SAM crée automatiquement un package de déploiement .zip ou une image de conteneur avec le code de votre fonction et les dépendances que vous spécifiez. Pour en savoir plus sur l'utilisation d'AWS SAM pour créer et déployer des fonctions Lambda, veuillez consulter la rubrique Premiers pas avec AWS SAM dans le Guide du développeur AWS Serverless Application Model.

Vous pouvez également utiliser AWS SAM pour créer une fonction Lambda à l'aide d'une archive de fichiers .zip existante. Pour créer une fonction Lambda à l'aide d'AWS SAM, vous pouvez enregistrer votre fichier .zip dans un compartiment Amazon S3 ou dans un dossier local sur votre machine de

génération. Pour savoir comment charger un fichier dans un compartiment Amazon S3 à l'aide de l'AWS CLI, veuillez consulter la rubrique Déplacer des objets dans le Guide de l'utilisateur AWS CLI.

Dans votre modèle AWS SAM, la ressource `AWS::Serverless::Function` spécifie votre fonction Lambda. Dans cette ressource, définissez les propriétés suivantes pour créer une fonction à l'aide d'une archive de fichiers .zip :

- `PackageType` : défini sur `Zip`
- `CodeUri` : défini comme l'URI Amazon S3 du code de la fonction, le chemin d'accès vers le dossier local ou l'objet FunctionCode
- `Runtime` : défini sur votre exécution choisie

Avec AWS SAM, si votre fichier .zip est supérieur à 50 Mo, vous n'avez pas à le charger au préalable dans un compartiment Amazon S3. AWS SAM peut charger des packages .zip jusqu'à la taille maximale autorisée de 250 Mo (décompressés) à partir d'un emplacement sur votre machine de génération locale.

Pour en savoir plus sur le déploiement de fonctions à l'aide d'un fichier .zip dans AWS SAM, veuillez consulter AWS::Serverless::Function dans le Guide du développeur AWS SAM.

Exemple : utilisation d'AWS SAM pour créer une fonction Go avec provided.al2023

1. Créez un modèle AWS SAM avec les propriétés suivantes :

   - BuildMethod : spécifie le compilateur de votre application. Utilisez `go1.x`.
   - Runtime : utilisez `provided.al2023`.
   - CodeUri : saisissez le chemin d'accès à votre code.
   - Architectures : Utilisez `[arm64]` pour l'architecture arm64. Pour l'architecture x86_64, utilisez `[amd64]` ou supprimez la propriété `Architectures`.

   Example template.yaml

   ```
   AWSTemplateFormatVersion: '2010-09-09'
   Transform: 'AWS::Serverless-2016-10-31'
   Resources:
     HelloWorldFunction:
       Type: AWS::Serverless::Function
       Metadata:
         BuildMethod: go1.x
   ```

```
    Properties:
      CodeUri: hello-world/ # folder where your main program resides
      Handler: bootstrap
      Runtime: provided.al2023
      Architectures: [arm64]
```

2. Utilisez la commande sam build pour compiler l'exécutable.

```
sam build
```

3. Utilisez la commande sam deploy pour déployer la fonction vers Lambda.

```
sam deploy --guided
```

Création et mise à jour de fonctions avec des fichiers .zip à l'aide d'CloudFormation

Vous pouvez utiliser CloudFormation pour créer une fonction Lambda à l'aide d'une archive de fichiers .zip. Pour créer une fonction Lambda à partir d'un fichier .zip, vous devez d'abord charger votre fichier dans un compartiment Amazon S3. Pour savoir comment charger un fichier dans un compartiment Amazon S3 à l'aide de l'AWS CLI, veuillez consulter la rubrique Déplacer des objets dans le Guide de l'utilisateur AWS CLI.

Dans votre modèle CloudFormation, la ressource `AWS::Lambda::Function` spécifie votre fonction Lambda. Dans cette ressource, définissez les propriétés suivantes pour créer une fonction à l'aide d'une archive de fichiers .zip :

- `PackageType` : défini sur `Zip`
- `Code` : saisissez le nom du compartiment Amazon S3 et le nom du fichier .zip dans les champs `S3Bucket` et `S3Key`
- `Runtime` : défini sur votre exécution choisie

Le fichier .zip que CloudFormation génère ne peut pas dépasser 4 Mo. Pour en savoir plus sur le déploiement de fonctions à l'aide d'un fichier .zip dans CloudFormation, veuillez consulter AWS::Lambda::Function dans le Guide de l'utilisateur CloudFormation.

Déployer des fonctions Lambda Go avec des images conteneurs

Il existe deux méthodes pour créer une image de conteneur pour une fonction Lambda Go :

- Utilisation d'une image de base AWS uniquement pour le système d'exploitation

  Go est implémenté différemment des autres exécutions gérées. Go se compilant nativement en un binaire exécutable, il n'a pas besoin d'un environnement d'exécution dédié au langage. Utilisez une image de base réservée au système d'exploitation pour créer des images Go pour Lambda. Pour rendre l'image compatible avec Lambda, vous devez inclure le package `aws-lambda-go/lambda` dans l'image.
- Utilisation d'une image non AWS basique

  Vous pouvez utiliser une autre image de base à partir d'un autre registre de conteneur, comme Alpine Linux ou Debian. Vous pouvez également utiliser une image personnalisée créée par votre organisation. Pour rendre l'image compatible avec Lambda, vous devez inclure le package `aws-lambda-go/lambda` dans l'image.

> **Tip**
>
> Pour réduire le temps nécessaire à l'activation des fonctions du conteneur Lambda, consultez Utiliser des générations en plusieurs étapes (français non garanti) dans la documentation Docker. Pour créer des images de conteneur efficaces, suivez la section Bonnes pratiques pour l'écriture de Dockerfiles (français non garanti).

Cette page explique comment créer, tester et déployer des images de conteneur pour Lambda.

AWS images de base pour le déploiement des fonctions Go

Go est implémenté différemment des autres exécutions gérées. Go se compilant nativement en un binaire exécutable, il n'a pas besoin d'un environnement d'exécution dédié au langage. Utilisez une image de base réservée au système d'exploitation pour déployer les fonctions Go sur Lambda.

Name	Identifiant	Système d'exploitation	Date d'obsolescence	Créer la fonction de blocage	Mettre à jour la fonction de blocage
Exécution réservée au système d'exploitation	`provided.al2023`	Amazon Linux 2	30 juin 2029	31 juillet 2029	31 août 2029
Exécution réservée au système d'exploitation	`provided.al2`	Amazon Linux 2	31 juillet 2026	31 août 2026	30 sept. 2026

Galerie publique d'Amazon Elastic Container Registry : gallery.ecr. aws/lambda/provided

Client d'interface d'environnement d'exécution Go

Le package `aws-lambda-go/lambda` inclut une implémentation de l'interface d'exécution. Pour obtenir des exemples d'utilisation de `aws-lambda-go/lambda` dans votre image, consultez Utilisation d'une image de base AWS uniquement pour le système d'exploitation ou Utilisation d'une image non AWS basique.

Utilisation d'une image de base AWS uniquement pour le système d'exploitation

Go est implémenté différemment des autres exécutions gérées. Go se compilant nativement en un binaire exécutable, il n'a pas besoin d'un environnement d'exécution dédié au langage. N'utilisez une image de base réservée au système d'exploitation que pour créer des images de conteneur pour les fonctions Go.

Étiquettes	Environnement d'exécution	Système d'exploitation	Dockerfile	Obsolescence
al2023	Exécution uniquement basée sur le système d'exploitation	Amazon Linux	Dockerfile pour OS uniquement Runtime sur GitHub	30 juin 2029
al2	Exécution uniquement basée sur le système d'exploitation	Amazon Linux 2	Dockerfile pour OS uniquement Runtime sur GitHub	31 juillet 2026

Pour plus d'informations sur ces images de base, consultez fourni située dans la galerie publique Amazon ECR.

Vous devez inclure le package aws-lambda-go/lambda dans votre gestionnaire Go. Ce package implémente le modèle de programmation pour Go, y compris l'interface d'exécution.

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- AWS CLI version 2
- Docker (version minimale 25.0.0)
- Le plug-in buildx Docker.
- Go

Création d'une image à partir de l'image de base provided.al2023

Pour créer et déployer une fonction Go avec l'image de base **`provided.al2023`**

1. Créez un répertoire pour le projet, puis passez à ce répertoire.

```
mkdir hello
cd hello
```

2. Initialisez un nouveau module Go.

```
go mod init example.com/hello-world
```

3. Ajoutez la bibliothèque lambda comme dépendance de votre nouveau module.

```
go get github.com/aws/aws-lambda-go/lambda
```

4. Créez un fichier nommé `main.go` et ouvrez-le dans un éditeur de texte. Il s'agit du code de la fonction Lambda. Vous pouvez utiliser l'exemple de code suivant pour le tester, ou le remplacer par le vôtre.

```
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, event events.APIGatewayProxyRequest)
 (events.APIGatewayProxyResponse, error) {
 response := events.APIGatewayProxyResponse{
  StatusCode: 200,
  Body:       "\"Hello from Lambda!\"",
 }
 return response, nil
}

func main() {
 lambda.Start(handler)
}
```

5. Utilisez un éditeur de texte afin de créer un Dockerfile dans votre répertoire de projets.

   - L'exemple de fichier Docker suivant utilise une génération en plusieurs étapes. Cela vous permet d'utiliser une image de base différente à chaque étape. Vous pouvez utiliser une image, telle qu'une image de base Go, pour compiler votre code et générer le binaire exécutable. Vous pouvez ensuite utiliser une image différente, telle que `provided.al2023`, dans l'instruction finale `FROM` pour définir l'image que vous déployez sur Lambda. Le processus de génération est séparé de l'image de déploiement final, de sorte que l'image finale ne contient que les fichiers nécessaires à l'exécution de l'application.
   - Vous pouvez utiliser la balise optionnelle `lambda.norpc` pour exclure le composant RPC (Remote Procedure Call) de la bibliothèque lambda. Le composant RPC n'est nécessaire que lorsque vous utilisez l'environnement d'exécution Go 1.x. L'exclusion du RPC réduit la taille du package de déploiement.
   - Notez que l'exemple de Dockerfile n'inclut pas d'instruction USER. Lorsque vous déployez une image de conteneur sur Lambda, Lambda définit automatiquement un utilisateur Linux par défaut disposant d'autorisations de moindre privilège. Ceci est différent du comportement standard de Docker qui est défini par défaut par l'utilisateur `root` lorsqu'aucune instruction `USER` n'est fournie.

   Example – Dockerfile de génération en plusieurs étapes

   > **Note**
   >
   > Assurez-vous que la version de Go que vous spécifiez dans votre Dockerfile (par exemple, `golang:1.20`) est la même que celle que vous avez utilisée pour créer votre application.

```
FROM golang:1.20 as build
WORKDIR /helloworld
# Copy dependencies list
COPY go.mod go.sum ./
# Build with optional lambda.norpc tag
COPY main.go .
RUN go build -tags lambda.norpc -o main main.go
# Copy artifacts to a clean image
FROM public.ecr.aws/lambda/provided:al2023
COPY --from=build /helloworld/main ./main
```

```
ENTRYPOINT [ "./main" ]
```

6. Générez l'image Docker à l'aide de la commande docker build. L'exemple suivant nomme l'image `docker-image` et lui donne la balise `test`. Pour rendre votre image compatible avec Lambda, vous devez utiliser l'option `--provenance=false`.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> La commande spécifie l'option `--platform linux/amd64` pour garantir la compatibilité de votre conteneur avec l'environnement d'exécution Lambda, quelle que soit l'architecture de votre machine de génération. Si vous avez l'intention de créer une fonction Lambda à l'aide de l'architecture du jeu ARM64 d'instructions, veillez à modifier la commande pour utiliser l'`--platform linux/arm64`option à la place.

(Facultatif) Testez l'image localement

Utilisez l'émulateur d'interface d'exécution pour tester votre image localement. L'émulateur d'interface d'exécution est inclus dans l'image de base `provided.al2023`.

Pour exécuter l'émulateur d'interface d'exécution sur votre ordinateur local

1. Démarrez votre image Docker à l'aide de la commande docker run. Notez ce qui suit :

   - `docker-image` est le nom de l'image et `test` est la balise.
   - `./main` est le ENTRYPOINT de votre Dockerfile.

```
docker run -d -p 9000:8080 \
--entrypoint /usr/local/bin/aws-lambda-rie \
docker-image:test ./main
```

   Cette commande exécute l'image en tant que conteneur et crée un point de terminaison local à `localhost:9000/2015-03-31/functions/function/invocations`.

2. À partir d'une nouvelle fenêtre de terminal, publiez un événement sur le point de terminaison suivant au moyen d'une commande curl :

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

Cette commande invoque la fonction avec un événement vide et renvoie une réponse. Certaines fonctions peuvent nécessiter l'ajout d'une charge utile JSON. Exemple :

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

3. Obtenez l'ID du conteneur.

```
docker ps
```

4. Utilisez la commande docker kill pour arrêter le conteneur. Dans cette commande, remplacez `3766c4ab331c` par l'ID du conteneur de l'étape précédente.

```
docker kill 3766c4ab331c
```

Déploiement de l'image

Pour charger l'image sur Amazon RIE et créer la fonction Lambda

1. Exécutez la get-login-passwordcommande pour authentifier la CLI Docker auprès de votre registre Amazon ECR.
   - Définissez la `--region` valeur à l' Région AWS endroit où vous souhaitez créer le référentiel Amazon ECR.
   - `111122223333`Remplacez-le par votre Compte AWS identifiant.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Créez un référentiel dans Amazon ECR à l'aide de la commande create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Le référentiel Amazon ECR doit être Région AWS identique à la fonction Lambda.

En cas de succès, vous obtenez une réponse comme celle-ci :

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Copiez le `repositoryUri` à partir de la sortie de l'étape précédente.
4. Exécutez la commande docker tag pour étiqueter votre image locale dans votre référentiel Amazon ECR en tant que dernière version. Dans cette commande :

   - `docker-image:test` est le nom et la balise de votre image Docker. Il s'agit du nom et de la balise de l'image que vous avez spécifiés dans la commande `docker build`.
   - Remplacez <ECRrepositoryUri> par l'`repositoryUri` que vous avez copié. Assurez-vous d'inclure `:latest` à la fin de l'URI.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Exemple :

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Exécutez la commande docker push pour déployer votre image locale dans le référentiel Amazon ECR. Assurez-vous d'inclure `:latest` à la fin de l'URI du référentiel.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Créez un rôle d'exécution pour la fonction, si vous n'en avez pas déjà un. Vous aurez besoin de l'Amazon Resource Name (ARN) du rôle à l'étape suivante.
7. Créez la fonction Lambda. Pour `ImageUri`, indiquez l'URI du référentiel mentionné précédemment. Assurez-vous d'inclure `:latest` à la fin de l'URI.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> Vous pouvez créer une fonction en utilisant une image d'un autre AWS compte, à condition que l'image se trouve dans la même région que la fonction Lambda. Pour de plus amples informations, veuillez consulter Autorisations entre comptes Amazon ECR.

8. Invoquer la fonction.

```
aws lambda invoke --function-name hello-world response.json
```

Vous devriez obtenir une réponse comme celle-ci :

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Pour voir la sortie de la fonction, consultez le fichier `response.json`.

Pour mettre à jour le code de fonction, vous devez créer à nouveau l'image, télécharger la nouvelle image dans le référentiel Amazon ECR, puis utiliser la update-function-codecommande pour déployer l'image sur la fonction Lambda.

Lambda résout l'étiquette d'image en hachage d'image spécifique. Cela signifie que si vous pointez la balise d'image qui a été utilisée pour déployer la fonction vers une nouvelle image dans Amazon ECR, Lambda ne met pas automatiquement à jour la fonction pour utiliser la nouvelle image.

Pour déployer la nouvelle image sur la même fonction Lambda, vous devez utiliser la update-function-codecommande, même si la balise d'image dans Amazon ECR reste la même. Dans l'exemple suivant, l'option `--publish` crée une version de la fonction à l'aide de l'image du conteneur mise à jour.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Utilisation d'une image non AWS basique

Vous pouvez créer une image de conteneur pour Go à partir d'une image non AWS basique. L'exemple de Dockerfile dans les étapes suivantes utilise une image de base Alpine.

Vous devez inclure le package aws-lambda-go/lambda dans votre gestionnaire Go. Ce package implémente le modèle de programmation pour Go, y compris l'interface d'exécution.

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- AWS CLI version 2
- Docker (version minimale 25.0.0)
- Le plug-in buildx Docker.
- Go

Création d'une image à partir d'une image de base alternative

Pour créer et déployer une fonction Go avec une image de base Alpine

1. Créez un répertoire pour le projet, puis passez à ce répertoire.

```
mkdir hello
cd hello
```

2. Initialisez un nouveau module Go.

```
go mod init example.com/hello-world
```

3. Ajoutez la bibliothèque lambda comme dépendance de votre nouveau module.

```
go get github.com/aws/aws-lambda-go/lambda
```

4. Créez un fichier nommé `main.go` et ouvrez-le dans un éditeur de texte. Il s'agit du code de la fonction Lambda. Vous pouvez utiliser l'exemple de code suivant pour le tester, ou le remplacer par le vôtre.

```
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, event events.APIGatewayProxyRequest)
 (events.APIGatewayProxyResponse, error) {
 response := events.APIGatewayProxyResponse{
  StatusCode: 200,
  Body:       "\"Hello from Lambda!\"",
 }
 return response, nil
}

func main() {
 lambda.Start(handler)
}
```

5. Utilisez un éditeur de texte afin de créer un Dockerfile dans votre répertoire de projets. L'exemple de Dockerfile suivant utilise une image de base Alpine. Notez que l'exemple de Dockerfile n'inclut pas d'instruction USER. Lorsque vous déployez une image de conteneur sur Lambda, Lambda définit automatiquement un utilisateur Linux par défaut disposant d'autorisations de moindre privilège. Ceci est différent du comportement standard de Docker qui est défini par défaut par l'utilisateur `root` lorsqu'aucune instruction USER n'est fournie.

   Example Dockerfile

   > **Note**
   >
   > Assurez-vous que la version de Go que vous spécifiez dans votre Dockerfile (par exemple, `golang:1.20`) est la même que celle que vous avez utilisée pour créer votre application.

   ```
   FROM golang:1.20.2-alpine3.16 as build
   WORKDIR /helloworld
   # Copy dependencies list
   COPY go.mod go.sum ./
   # Build
   COPY main.go .
   RUN go build -o main main.go
   # Copy artifacts to a clean image
   FROM alpine:3.16
   COPY --from=build /helloworld/main /main
   ENTRYPOINT [ "/main" ]
   ```

6. Générez l'image Docker à l'aide de la commande docker build. L'exemple suivant nomme l'image `docker-image` et lui donne la balise `test`. Pour rendre votre image compatible avec Lambda, vous devez utiliser l'option `--provenance=false`.

   ```
   docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
    .
   ```

   > **Note**
   >
   > La commande spécifie l'option `--platform linux/amd64` pour garantir la compatibilité de votre conteneur avec l'environnement d'exécution Lambda, quelle que

soit l'architecture de votre machine de génération. Si vous avez l'intention de créer une fonction Lambda à l'aide de l'architecture du jeu ARM64 d'instructions, veillez à modifier la commande pour utiliser l'`--platform linux/arm64`option à la place.

(Facultatif) Testez l'image localement

Utilisez l'émulateur d'interface d'exécution pour tester l'image localement. Vous pouvez intégrer l'émulateur dans votre image ou utiliser la procédure suivante pour l'installer sur votre machine locale.

Pour installer et exécuter l'émulateur d'interface d'exécution sur votre ordinateur local

1. Depuis le répertoire de votre projet, exécutez la commande suivante pour télécharger l'émulateur d'interface d'exécution (architecture x86-64) GitHub et l'installer sur votre machine locale.

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   Pour installer l'émulateur arm64, remplacez l'URL du GitHub référentiel dans la commande précédente par la suivante :

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

   PowerShell

   ```
   $dirPath = "$HOME\.aws-lambda-rie"
   if (-not (Test-Path $dirPath)) {
       New-Item -Path $dirPath -ItemType Directory
   }

   $downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie"
   $destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
   Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
   ```

Pour installer l'émulateur arm64, remplacez `$downloadLink` par ce qui suit :

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. Démarrez votre image Docker à l'aide de la commande docker run. Notez ce qui suit :

   - `docker-image` est le nom de l'image et `test` est la balise.
   - `/main` est le `ENTRYPOINT` de votre Dockerfile.

   Linux/macOS

   ```
   docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
    9000:8080 \
       --entrypoint /aws-lambda/aws-lambda-rie \
       docker-image:test \
           /main
   ```

   PowerShell

   ```
   docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
    9000:8080 `
   --entrypoint /aws-lambda/aws-lambda-rie `
   docker-image:test `
       /main
   ```

   Cette commande exécute l'image en tant que conteneur et crée un point de terminaison local à `localhost:9000/2015-03-31/functions/function/invocations`.

   > **Note**
   >
   > Si vous avez créé l'image Docker pour l'architecture du jeu ARM64 d'instructions, veillez à utiliser l'`--platform linux/arm64`option au lieu de`--platform linux/amd64`.

3. Publiez un événement au point de terminaison local.

Linux/macOS

Sous Linux et macOS, exécutez la commande `curl` suivante :

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

Cette commande invoque la fonction avec un événement vide et renvoie une réponse. Si vous utilisez votre propre code de fonction plutôt que l'exemple de code de fonction, vous pouvez invoquer la fonction avec une charge utile JSON. Exemple :

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

Dans PowerShell, exécutez la `Invoke-WebRequest` commande suivante :

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Cette commande invoque la fonction avec un événement vide et renvoie une réponse. Si vous utilisez votre propre code de fonction plutôt que l'exemple de code de fonction, vous pouvez invoquer la fonction avec une charge utile JSON. Exemple :

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. Obtenez l'ID du conteneur.

```
docker ps
```

5. Utilisez la commande docker kill pour arrêter le conteneur. Dans cette commande, remplacez `3766c4ab331c` par l'ID du conteneur de l'étape précédente.

```
docker kill 3766c4ab331c
```

Déploiement de l'image

Pour charger l'image sur Amazon RIE et créer la fonction Lambda

1. Exécutez la get-login-passwordcommande pour authentifier la CLI Docker auprès de votre registre Amazon ECR.

   - Définissez la `--region` valeur à l' Région AWS endroit où vous souhaitez créer le référentiel Amazon ECR.
   - `111122223333`Remplacez-le par votre Compte AWS identifiant.

   ```
   aws ecr get-login-password --region us-east-1 | docker login --username AWS --
   password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
   ```

2. Créez un référentiel dans Amazon ECR à l'aide de la commande create-repository.

   ```
   aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
   scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
   ```

   > **Note**
   >
   > Le référentiel Amazon ECR doit être Région AWS identique à la fonction Lambda.

   En cas de succès, vous obtenez une réponse comme celle-ci :

   ```
   {
       "repository": {
           "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
   world",
           "registryId": "111122223333",
           "repositoryName": "hello-world",
           "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
   world",
           "createdAt": "2023-03-09T10:39:01+00:00",
           "imageTagMutability": "MUTABLE",
           "imageScanningConfiguration": {
               "scanOnPush": true
           },
           "encryptionConfiguration": {
   ```

```
            "encryptionType": "AES256"
        }
    }
}
```

3. Copiez le `repositoryUri` à partir de la sortie de l'étape précédente.
4. Exécutez la commande docker tag pour étiqueter votre image locale dans votre référentiel Amazon ECR en tant que dernière version. Dans cette commande :
   - `docker-image:test` est le nom et la balise de votre image Docker. Il s'agit du nom et de la balise de l'image que vous avez spécifiés dans la commande `docker build`.
   - Remplacez <ECRrepositoryUri> par l'`repositoryUri` que vous avez copié. Assurez-vous d'inclure `:latest` à la fin de l'URI.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

   Exemple :

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Exécutez la commande docker push pour déployer votre image locale dans le référentiel Amazon ECR. Assurez-vous d'inclure `:latest` à la fin de l'URI du référentiel.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Créez un rôle d'exécution pour la fonction, si vous n'en avez pas déjà un. Vous aurez besoin de l'Amazon Resource Name (ARN) du rôle à l'étape suivante.
7. Créez la fonction Lambda. Pour `ImageUri`, indiquez l'URI du référentiel mentionné précédemment. Assurez-vous d'inclure `:latest` à la fin de l'URI.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> Vous pouvez créer une fonction en utilisant une image d'un autre AWS compte, à condition que l'image se trouve dans la même région que la fonction Lambda. Pour de plus amples informations, veuillez consulter Autorisations entre comptes Amazon ECR.

8. Invoquer la fonction.

```
aws lambda invoke --function-name hello-world response.json
```

Vous devriez obtenir une réponse comme celle-ci :

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Pour voir la sortie de la fonction, consultez le fichier `response.json`.

Pour mettre à jour le code de fonction, vous devez créer à nouveau l'image, télécharger la nouvelle image dans le référentiel Amazon ECR, puis utiliser la update-function-codecommande pour déployer l'image sur la fonction Lambda.

Lambda résout l'étiquette d'image en hachage d'image spécifique. Cela signifie que si vous pointez la balise d'image qui a été utilisée pour déployer la fonction vers une nouvelle image dans Amazon ECR, Lambda ne met pas automatiquement à jour la fonction pour utiliser la nouvelle image.

Pour déployer la nouvelle image sur la même fonction Lambda, vous devez utiliser la update-function-codecommande, même si la balise d'image dans Amazon ECR reste la même. Dans l'exemple suivant, l'option `--publish` crée une version de la fonction à l'aide de l'image du conteneur mise à jour.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Utilisation de couches pour les fonctions Lambda Go

Nous déconseillons d'utiliser des couches pour gérer les dépendances des fonctions Lambda écrites en Go. Cela est dû au fait que les fonctions Lambda en Go sont compilées en un seul exécutable, que vous fournissez à Lambda lorsque vous déployez votre fonction. Cet exécutable contient votre code de fonction compilé, ainsi que toutes ses dépendances. L'utilisation de couches complique non seulement ce processus, mais entraîne également une augmentation des temps de démarrage à froid, car vos fonctions doivent charger manuellement des assemblages supplémentaires en mémoire pendant la phase d'initialisation.

Pour utiliser des dépendances externes avec vos gestionnaires Go, incluez-les directement dans votre package de déploiement. Ce faisant, vous simplifiez le processus de déploiement et profitez des optimisations du compilateur Go intégré. Pour un exemple d'importation et d'utilisation d'une dépendance telle que le kit SDK AWS pour Go dans votre fonction, consultez the section called "Handler (Gestionnaire)".

Journalisation et surveillance des fonctions Lambda Go

AWS Lambda surveille automatiquement les fonctions Lambda en votre nom et envoie les journaux à Amazon. CloudWatch Votre fonction Lambda est fournie avec un groupe de CloudWatch journaux Logs et un flux de journaux pour chaque instance de votre fonction. L'environnement d'exécution Lambda envoie des informations sur chaque invocation au flux de journaux et transmet les journaux et autres sorties provenant du code de votre fonction. Pour de plus amples informations, veuillez consulter Envoi automatique des journaux de la fonction Lambda à CloudWatch Logs.

Cette page explique comment générer une sortie de journal à partir du code de votre fonction Lambda et comment accéder aux journaux à l'aide de la AWS Command Line Interface console Lambda ou de la console. CloudWatch

Sections

- Création d'une fonction qui renvoie des journaux
- Affichage des journaux dans la console Lambda
- Afficher les journaux dans la CloudWatch console
- Afficher les journaux à l'aide de AWS Command Line Interface (AWS CLI)
- Suppression de journaux

Création d'une fonction qui renvoie des journaux

Pour générer les journaux à partir de votre code de fonction, vous pouvez utiliser des méthodes sur le package fmt ou n'importe quelle bibliothèque de journalisation qui écrit dans `stdout` ou `stderr`. L'exemple suivant utilise le package de journal.

Example main.go – journalisation

```
func handleRequest(ctx context.Context, event events.SQSEvent) (string, error) {
  // event
  eventJson, _ := json.MarshalIndent(event, "", "  ")
  log.Printf("EVENT: %s", eventJson)
  // environment variables
  log.Printf("REGION: %s", os.Getenv("AWS_REGION"))
  log.Println("ALL ENV VARS:")
  for _, element := range os.Environ() {
    log.Println(element)
```

```
  }
```

Example format des journaux

```
START RequestId: dbda340c-xmpl-4031-8810-11bb609b4c71 Version: $LATEST
2020/03/27 03:40:05 EVENT: {
  "Records": [
    {
      "messageId": "19dd0b57-b21e-4ac1-bd88-01bbb068cb78",
      "receiptHandle": "MessageReceiptHandle",
      "body": "Hello from SQS!",
      "md5OfBody": "7b27xmplb47ff90a553787216d55d91d",
      "md5OfMessageAttributes": "",
      "attributes": {
        "ApproximateFirstReceiveTimestamp": "1523232000001",
        "ApproximateReceiveCount": "1",
        "SenderId": "123456789012",
        "SentTimestamp": "1523232000000"
      },
      ...
2020/03/27 03:40:05 AWS_LAMBDA_LOG_STREAM_NAME=2020/03/27/
[$LATEST]569cxmplc3c34c7489e6a97ad08b4419
2020/03/27 03:40:05 AWS_LAMBDA_FUNCTION_NAME=blank-go-function-9DV3XMPL6XBC
2020/03/27 03:40:05 AWS_LAMBDA_FUNCTION_MEMORY_SIZE=128
2020/03/27 03:40:05 AWS_LAMBDA_FUNCTION_VERSION=$LATEST
2020/03/27 03:40:05 AWS_EXECUTION_ENV=AWS_Lambda_go1.x
END RequestId: dbda340c-xmpl-4031-8810-11bb609b4c71
REPORT RequestId: dbda340c-xmpl-4031-8810-11bb609b4c71 Duration: 38.66 ms Billed
 Duration: 243 ms Memory Size: 128 MB Max Memory Used: 54 MB Init Duration: 203.69 ms
XRAY TraceId: 1-5e7d7595-212fxmpl9ee07c4884191322 SegmentId: 42ffxmpl0645f474 Sampled:
 true
```

L’environnement d’exécution Go enregistre les lignes `START`, `END` et `REPORT` pour chaque invocation. La ligne de rapport fournit les détails suivants.

Champs de données de la ligne REPORT

- RequestId— L'identifiant de demande unique pour l'invocation.
- Duration – Temps que la méthode de gestion du gestionnaire de votre fonction a consacré au traitement de l’événement.
- Billed Duration : temps facturé pour l’invocation.

- Memory Size – Quantité de mémoire allouée à la fonction.
- Max Memory Used – Quantité de mémoire utilisée par la fonction. Lorsque les appels partagent un environnement d'exécution, Lambda indique la mémoire maximale utilisée pour toutes les invocations. Ce comportement peut entraîner une valeur signalée plus élevée que prévu.
- Init Duration : pour la première requête servie, temps qu'il a pris à l'exécution charger la fonction et exécuter le code en dehors de la méthode du gestionnaire.
- XRAY TraceId — Pour les demandes suivies, l'ID de AWS X-Ray trace.
- SegmentId— Pour les demandes tracées, l'identifiant du segment X-Ray.
- Sampled – Pour les demandes suivies, résultat de l'échantillonnage.

Affichage des journaux dans la console Lambda

Vous pouvez utiliser la console Lambda pour afficher la sortie du journal après avoir invoqué une fonction Lambda.

Si votre code peut être testé à partir de l'éditeur Code intégré, vous trouverez les journaux dans les résultats d'exécution. Lorsque vous utilisez la fonctionnalité de test de console pour invoquer une fonction, vous trouverez Sortie du journal dans la section Détails.

Afficher les journaux dans la CloudWatch console

Vous pouvez utiliser la CloudWatch console Amazon pour consulter les journaux de toutes les invocations de fonctions Lambda.

Pour afficher les journaux sur la CloudWatch console

1. Ouvrez la page Groupes de journaux sur la CloudWatch console.
2. Choisissez le groupe de journaux pour votre fonction (***your-function-name***/aws/lambda/).
3. Choisissez un flux de journaux.

Chaque flux de journal correspond à une instance de votre fonction. Un flux de journal apparaît lorsque vous mettez à jour votre fonction Lambda et lorsque des instances supplémentaires sont créées pour gérer les appels simultanés. Pour trouver les journaux d'un appel spécifique, nous vous recommandons d'instrumenter votre fonction avec. AWS X-Ray X-Ray enregistre des détails sur la demande et le flux de journaux dans le suivi.

Afficher les journaux à l'aide de AWS Command Line Interface (AWS CLI)

AWS CLI Il s'agit d'un outil open source qui vous permet d'interagir avec les AWS services à l'aide de commandes dans votre interface de ligne de commande. Pour effectuer les étapes de cette section, vous devez disposer de la version 2 de l'AWS CLI.

Vous pouvez utiliser AWS CLI pour récupérer les journaux d'une invocation à l'aide de l'option de commande `--log-type`. La réponse inclut un champ `LogResult` qui contient jusqu'à 4 Ko de journaux codés en base64 provenant de l'invocation.

Example récupérer un ID de journal

L'exemple suivant montre comment récupérer un ID de journal à partir du champ `LogResult` d'une fonction nommée `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Vous devriez voir la sortie suivante:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example décoder les journaux

Dans la même invite de commandes, utilisez l'utilitaire `base64` pour décoder les journaux. L'exemple suivant montre comment récupérer les journaux encodés en base64 pour `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

Vous devriez voir la sortie suivante :

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

L'utilitaire `base64` est disponible sous Linux, macOS et Ubuntu sous Windows. Les utilisateurs de macOS auront peut-être besoin d'utiliser `base64 -D`.

Example Script get-logs.sh

Dans la même invite de commandes, utilisez le script suivant pour télécharger les cinq derniers événements de journalisation. Le script utilise `sed` pour supprimer les guillemets du fichier de sortie et attend 15 secondes pour permettre la mise à disposition des journaux. La sortie comprend la réponse de Lambda, ainsi que la sortie de la commande `get-log-events`.

Copiez le contenu de l'exemple de code suivant et enregistrez-le dans votre répertoire de projet Lambda sous `get-logs.sh`.

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS et Linux (uniquement)

Dans la même invite de commandes, les utilisateurs macOS et Linux peuvent avoir besoin d'exécuter la commande suivante pour s'assurer que le script est exécutable.

```
chmod -R 755 get-logs.sh
```

Example récupérer les cinq derniers événements de journal

Dans la même invite de commande, exécutez le script suivant pour obtenir les cinq derniers événements de journalisation.

```
./get-logs.sh
```

Vous devriez voir la sortie suivante:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
```

```
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


Suppression de journaux

Les groupes de journaux ne sont pas supprimés automatiquement quand vous supprimez une fonction. Pour éviter de stocker des journaux indéfiniment, supprimez le groupe de journaux ou configurez une période de conservation à l'issue de laquelle les journaux sont supprimés automatiquement.

Instrumentation du code Go dans AWS Lambda

Lambda s'intègre avec AWS X-Ray pour vous aider à suivre, déboguer et optimiser les applications Lambda. Vous pouvez utiliser X-Ray pour suivre une demande lorsque celle-ci parcourt les ressources de votre application, qui peuvent inclure des fonctions Lambda et d'autres services AWS.

Pour envoyer des données de suivi à X-Ray, vous pouvez utiliser l'une des deux bibliothèques SDK suivantes :

- AWS Distro de OpenTelemetry (ADOT) : solution sécurisée, prête pour la production, distribution prise en charge par AWS du kit SDK OpenTelemetry (OTel).
- AWS X-Ray SDK pour Go Un kit SDK permettant de générer et d'envoyer des données de suivi par rayons X.

Chacun des kits SDK offre des moyens d'envoyer vos données de télémétrie au service X-Ray. Vous pouvez ensuite utiliser X-Ray pour afficher, filtrer et avoir un aperçu des métriques de performance de votre application, afin d'identifier les problèmes et les occasions d'optimiser votre application.

> **Important**
>
> X-Ray et Powertools pour les kits SDK AWS Lambda font partie d'une solution d'instrumentation étroitement intégrée proposée par AWS. Les couches ADOT Lambda font partie d'une norme industrielle pour l'instrumentation de traçage qui collecte plus de données en général, mais qui peut ne pas convenir à tous les cas d'utilisation. Vous pouvez mettre en œuvre le traçage de bout en bout dans X-Ray en utilisant l'une ou l'autre solution. Pour en savoir plus sur le choix entre les deux, consultez Choix entre les kits SDK AWS Distro for OpenTelemetry et X-Ray (français non garanti).

Sections

- Utilisation d'ADOT pour instrumenter vos fonctions Go
- Utilisation du kit SDK X-Ray pour instrumenter vos fonctions Go
- Activation du suivi avec la console Lambda
- Activation du suivi avec l'API Lambda
- Activation du suivi avec CloudFormation
- Interprétation d'un suivi X-Ray

Utilisation d'ADOT pour instrumenter vos fonctions Go

ADOT fournit des couches Lambda entièrement gérées qui regroupent tout ce dont vous avez besoin pour collecter des données de télémétrie à l'aide du kit SDK OTel. En consommant cette couche, vous pouvez instrumenter vos fonctions Lambda sans avoir à modifier le code de fonction. Vous pouvez également configurer votre couche pour effectuer une initialisation personnalisée d'OTel. Pour de plus amples informations, veuillez consulter Configuration personnalisée pour ADOT Collector sur Lambda dans la documentation ADOT.

Pour les exécutions python, vous pouvez ajouter la AWS couche Lambda gérée pour ADOT Go pour instrumenter automatiquement vos fonctions. Pour obtenir des instructions détaillées sur la façon d'ajouter cette couche, veuillez consulter AWS Distro pour prise en charge d'OpenTelemetry Lambda pour Go dans la documentation ADOT.

Utilisation du kit SDK X-Ray pour instrumenter vos fonctions Go

Pour enregistrer des détails sur les appels effectués par votre fonction Lambda à d'autres ressources de votre application, vous pouvez également utiliser le SDK AWS X-Ray pour Go. Téléchargez le kit de développement logiciel (SDK) à partir de son référentiel GitHub avec `go get` :

```
go get github.com/aws/aws-xray-sdk-go
```

Pour instrumenter les clients AWS SDK, envoyez le client vers la méthode `xray.AWS()`. Ensuite, vous pouvez suivre vos appels à l'aide de la version `WithContext` de la méthode.

```
svc := s3.New(session.New())
xray.AWS(svc.Client)
...
svc.ListBucketsWithContext(ctx aws.Context, input *ListBucketsInput)
```

Une fois que vous avez ajouté les bonnes dépendances et effectué les modifications de code nécessaires, activez le suivi dans la configuration de votre fonction via la console Lambda ou l'API.

Activation du suivi avec la console Lambda

Pour activer/désactiver le traçage actif sur votre fonction Lambda avec la console, procédez comme suit :

Pour activer le traçage actif

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Choisissez Configuration, puis choisissez Outils de surveillance et d'opérations.
4. Dans Outils de surveillance supplémentaires, choisissez Modifier.
5. Sous CloudWatch Application Signals et AWS X-Ray sélectionnez Activer pour Suivis de service Lambda.
6. Choisissez Enregistrer.

Activation du suivi avec l'API Lambda

Configurez le suivi sur votre fonction Lambda avec le AWS CLI ou le kit SDK AWS, utilisez les opérations d'API suivantes :

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

L'exemple de commande AWS CLI suivant active le suivi actif sur une fonction nommée my-function.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Le mode de suivi fait partie de la configuration spécifique de la version lorsque vous publiez une version de votre fonction. Vous ne pouvez pas modifier le mode de suivi sur une version publiée.

Activation du suivi avec CloudFormation

Pour activer le suivi d'une ressource `AWS::Lambda::Function` dans un modèle CloudFormation, utilisez la propriété `TracingConfig`.

Example function-inline.yml – Configuration du suivi

```
Resources:
```

```
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

Pour une ressource AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function`, utilisez la propriété `Tracing`.

Example template.yml – Configuration du suivi

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

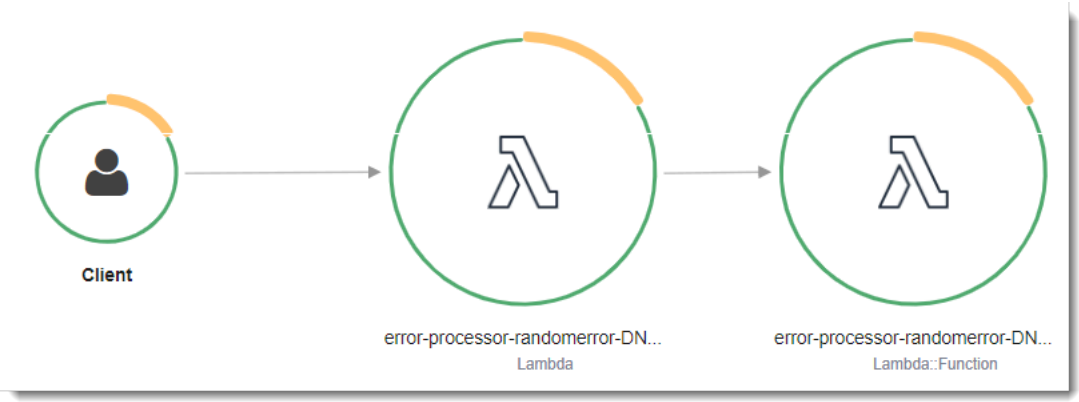
Interprétation d'un suivi X-Ray

Votre fonction a besoin d'une autorisation pour charger des données de suivi vers X-Ray. Lorsque vous activez le suivi actif dans la console Lambda, Lambda ajoute les autorisations requises au rôle d'exécution de votre fonction. Sinon, ajoutez la politique AWSXRayDaemonWriteAccess au rôle d'exécution.

Une fois que vous avez configuré le suivi actif, vous pouvez observer des demandes spécifiques via votre application. Le graphique de services X-Ray affiche des informations sur votre application et tous ses composants. L'exemple suivant montre une application dotée de deux fonctions. La fonction principale traite les événements et renvoie parfois des erreurs. La deuxième fonction en partant du haut traite les erreurs qui apparaissent dans le groupe de journaux du premier et utilise le kit SDK AWS pour appeler X-Ray, Amazon Simple Storage Service (Amazon S3) et Amazon CloudWatch logs.

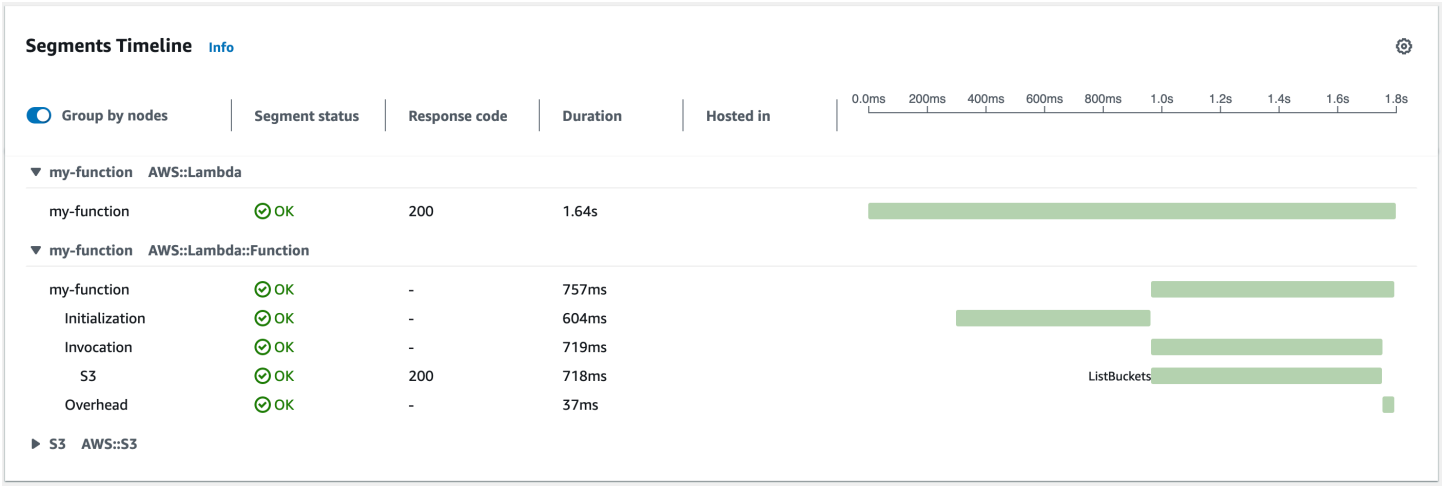
X-Ray ne trace pas toutes les requêtes vers votre application. X-Ray applique un algorithme d'échantillonnage pour s'assurer que le suivi est efficace, tout en fournissant un échantillon représentatif de toutes les demandes. Le taux d'échantillonnage est 1 demande par seconde et 5 % de demandes supplémentaires. Vous ne pouvez pas configurer ce taux d'échantillonnage X-Ray pour vos fonctions.

Dans X-Ray, un suivi enregistre des informations sur une demande traitée par un ou plusieurs services. Lambda enregistre deux segments par suivi, ce qui a pour effet de créer deux nœuds sur le graphique du service. L'image suivante met en évidence ces deux nœuds :



Le premier nœud sur la gauche représente le service Lambda qui reçoit la demande d'invocation. Le deuxième nœud représente votre fonction Lambda spécifique. L'exemple suivant illustre une trace avec ces deux segments. Les deux sont nommés my-function, mais l'un a pour origine `AWS::Lambda` et l'autre a pour origine `AWS::Lambda::Function`. Si le segment `AWS::Lambda` affiche une erreur, cela signifie que le service Lambda a rencontré un problème. Si le segment

`AWS::Lambda::Function` affiche une erreur, cela signifie que votre fonction a rencontré un problème.



Cet exemple développe le segment `AWS::Lambda::Function` pour afficher ses trois sous-segments.

> **Note**
>
> AWS est en train de modifier le service Lambda. En raison de ces modifications, vous pouvez constater des différences mineures entre la structure et le contenu des messages du journal système et des segments de suivi émis par les différentes fonctions Lambda de votre Compte AWS.
> L'exemple de suivi présenté ici illustre le segment de fonction à l'ancienne. Les différences entre les segments à l'ancienne et de style moderne sont décrites dans les paragraphes suivants.
> Ces modifications seront mises en œuvre au cours des prochaines semaines, et toutes les fonctions dans toutes les Régions AWS, sauf les régions de Chine et GovCloud, passeront à l'utilisation du nouveau format des messages de journal et des segments de suivi.

Le segment de fonction à l'ancienne contient les sous-segments suivants :

- Initialization (Initialisation) : représente le temps passé à charger votre fonction et à exécuter le code d'initialisation. Ce sous-segment apparaît pour le premier événement traité par chaque instance de votre fonction.
- Invocation – Représente le temps passé à exécuter votre code de gestionnaire.

- Overhead (Travail supplémentaire) – Représente le temps que le fichier d'exécution Lambda passe à se préparer à gérer l'événement suivant.

Le segment de fonction de style moderne ne contient pas de sous-segment `Invocation`. À la place, les sous-segments du client sont directement rattachés au segment de fonction. Pour plus d'informations sur la structure des segments de fonction à l'ancienne et de style moderne, consultez the section called "Comprendre les suivis X-Ray".

Vous pouvez également utiliser des clients HTTP, enregistrer des requêtes SQL et créer des sous-segments personnalisés avec des annotations et des métadonnées. Pour plus d'informations, consultez AWS X-RayKit SDK X-Ray pour Go dans le AWS X-RayGuide du développeur.

> **Tarification**
>
> Vous pouvez utiliser gratuitement le traçage par X-Ray chaque mois jusqu'à une certaine limite dans le cadre de l'offre gratuite AWS. Au-delà de ce seuil, X-Ray facture le stockage et la récupération du suivi. Pour en savoir plus, consultez Tarification de AWS X-Ray.

Création de fonctions Lambda avec C#

Vous pouvez exécuter votre application .NET dans Lambda à l'aide du moteur d'exécution .NET 8 géré, d'un environnement d'exécution personnalisé ou d'une image de conteneur. Une fois le code de votre application compilé, vous pouvez le déployer dans Lambda sous la forme d'un fichier .zip ou d'une image de conteneur. Lambda fournit les exécutions suivantes pour les langages .NET :

Name	Identifiant	Système d'exploitation	Date d'obsolescence	Créer la fonction de blocage	Mettre à jour la fonction de blocage
.NET 10	`dotnet10`	Amazon Linux 2	14 novembre 2028	14 déc. 2028	15 janvier 2029
.NET 9 (conteneur uniquement)	`dotnet9`	Amazon Linux 2	10 novembre 2026	Non planifié	Non planifié
.NET 8	`dotnet8`	Amazon Linux 2	10 novembre 2026	10 déc. 2026	11 janvier 2027

Configuration de votre environnement de développement .NET

Pour développer et créer vos fonctions Lambda, vous pouvez utiliser n'importe quel environnement de développement intégré .NET couramment disponible (IDEs), notamment Microsoft Visual Studio, Visual Studio Code et JetBrains Rider. Pour simplifier votre expérience de développement, AWS fournit un ensemble de modèles de projets .NET, ainsi que l'interface de ligne de `Amazon.Lambda.Tools` commande (CLI).

Exécutez les commandes CLI .NET suivantes pour installer ces modèles de projet et ces outils de ligne de commande.

Installation des modèles de projet .NET

Pour installer les modèles de projet, exécutez la commande suivante :

```
dotnet new install Amazon.Lambda.Templates
```

Installation et mise à jour des outils CLI

Exécutez les commandes suivantes pour installer, mettre à jour ou désinstaller la CLI `Amazon.Lambda.Tools`.

Pour installer les outils de ligne de commande :

```
dotnet tool install -g Amazon.Lambda.Tools
```

Pour mettre à jour les outils de ligne de commande :

```
dotnet tool update -g Amazon.Lambda.Tools
```

Pour désinstaller les outils de ligne de commande :

```
dotnet tool uninstall -g Amazon.Lambda.Tools
```

Définition du gestionnaire de fonction Lambda dans C#

Le gestionnaire de fonction Lambda est la méthode dans votre code de fonction qui traite les événements. Lorsque votre fonction est invoquée, Lambda exécute la méthode du gestionnaire. Votre fonction s'exécute jusqu'à ce que le gestionnaire renvoie une réponse, se ferme ou expire.

Cette page explique comment utiliser les gestionnaires de fonctions Lambda dans C# pour travailler avec l'environnement d'exécution géré .NET, notamment des options pour la configuration du projet, les conventions de dénomination et les pratiques exemplaires. Cette page inclut également un exemple de fonction Lambda C# qui collecte des informations relatives à une commande, produit un reçu sous forme de fichier texte et place ce fichier dans un compartiment Amazon Simple Storage Service (S3). Pour plus d'informations sur le déploiement de votre fonction après l'avoir écrite, consultez the section called "Package de déploiement" ou the section called "Déploiement d'images de conteneur".

Rubriques

- Configuration de votre projet de gestionnaire C#
- Exemple de fonction Lambda C#
- Gestionnaires de bibliothèques de classes
- Gestionnaires d'assemblages exécutables
- Signatures de gestionnaire valides pour les fonctions C#
- Convention de nommage du gestionnaire
- Sérialisation dans les fonctions Lambda C#
- Fonctions basées sur des fichiers
- Accès et utilisation de l'objet de contexte Lambda
- Utilisation de la SDK pour .NET v3 dans votre gestionnaire
- Accès aux variables d'environnement
- Utilisation de l'état global
- Simplifiez le code de la fonction à l'aide du cadre d'annotations Lambda
- Pratiques exemplaires de codage pour les fonctions Lambda C#

Configuration de votre projet de gestionnaire C#

Lorsque vous utilisez des fonctions Lambda en C#, le processus consiste à écrire votre code, puis à le déployer sur Lambda. Il existe deux modèles de déploiement différents pour les fonctions Lambda dans .NET : l'approche de la bibliothèque de classes et l'approche de l'assemblage exécutable.

Dans l'approche de bibliothèque de classes, vous empaquetez le code de votre fonction sous forme d'assemblage .NET (`.dll`) et le déployez sur Lambda avec l'environnement d'exécution géré .NET (`dotnet8`). Pour le nom du gestionnaire, Lambda attend une chaîne au format `AssemblyName::Namespace.Classname::Methodname`. Pendant la phase d'initialisation de la fonction, la classe de votre fonction est initialisée et tout code contenu dans le constructeur est exécuté.

Dans l'approche d'assemblage exécutable, vous utilisez la fonctionnalité d'instructions de haut niveau qui a été introduite pour la première fois avec C# 9. Cette approche génère un assemblage exécutable que Lambda exécute chaque fois qu'il reçoit une commande d'invocation pour votre fonction. Dans cette approche, vous utilisez également l'environnement d'exécution géré .NET (`dotnet8`). Pour le nom de gestionnaire, vous fournissez à Lambda le nom de l'assemblage exécutable à exécuter.

L'exemple principal de cette page illustre l'approche de bibliothèque de classes. Vous pouvez initialiser votre projet Lambda C# de différentes manières, mais la méthode la plus simple consiste à utiliser la CLI .NET avec `Amazon.Lambda.Tools` CLI. Configurez `Amazon.Lambda.Tools` CLI en suivant les étapes décrites dans the section called "Environnement de développement". Ensuite, initialisez votre projet avec la commande suivante :

```
dotnet new lambda.EmptyFunction --name ExampleCS
```

Cette commande génère la structure de fichiers suivante :

```
/project-root
    # src
        # ExampleCS
            # Function.cs (contains main handler)
            # Readme.md
            # aws-lambda-tools-defaults.json
            # ExampleCS.csproj
    # test
         # ExampleCS.Tests
            # FunctionTest.cs (contains main handler)
```

```
            # ExampleCS.Tests.csproj
```

Dans cette structure de fichiers, la logique du gestionnaire principal de votre fonction réside dans le fichier `Function.cs`.

Exemple de fonction Lambda C#

L'exemple de code de fonction Lambda C# suivant prend en compte les informations relatives à une commande, produit un reçu sous forme de fichier texte et place ce fichier dans un compartiment Amazon S3.

Example Fonction Lambda **`Function.cs`**

```
using System;
using System.Text;
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.S3;
using Amazon.S3.Model;

// Assembly attribute to enable Lambda function logging
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace ExampleLambda;

public class Order
{
    public string OrderId { get; set; } = string.Empty;
    public double Amount { get; set; }
    public string Item { get; set; } = string.Empty;
}

public class OrderHandler
{
    private static readonly AmazonS3Client s3Client = new();

    public async Task<string> HandleRequest(Order order, ILambdaContext context)
    {
        try
        {
            string? bucketName = Environment.GetEnvironmentVariable("RECEIPT_BUCKET");
            if (string.IsNullOrWhiteSpace(bucketName))
```

```
            {
                throw new ArgumentException("RECEIPT_BUCKET environment variable is not
 set");
            }

            string receiptContent = $"OrderID: {order.OrderId}\nAmount:
 ${order.Amount:F2}\nItem: {order.Item}";
            string key = $"receipts/{order.OrderId}.txt";

            await UploadReceiptToS3(bucketName, key, receiptContent);

            context.Logger.LogInformation($"Successfully processed order
 {order.OrderId} and stored receipt in S3 bucket {bucketName}");
            return "Success";
        }
        catch (Exception ex)
        {
            context.Logger.LogError($"Failed to process order: {ex.Message}");
            throw;
        }
    }

    private async Task UploadReceiptToS3(string bucketName, string key, string
 receiptContent)
    {
        try
        {
            var putRequest = new PutObjectRequest
            {
                BucketName = bucketName,
                Key = key,
                ContentBody = receiptContent,
                ContentType = "text/plain"
            };

            await s3Client.PutObjectAsync(putRequest);
        }
        catch (AmazonS3Exception ex)
        {
            throw new Exception($"Failed to upload receipt to S3: {ex.Message}", ex);
        }
    }
}
```

Ce fichier `Function.cs` comprend les sections suivantes :

- Instructions `using` : utilisez ces instructions pour importer les classes C# que votre fonction Lambda nécessite.
- `[assembly: LambdaSerializer(...)]` : `LambdaSerializer` est un attribut d'assemblage qui indique à Lambda de convertir automatiquement les données utiles des événements JSON en objets C# avant de les transmettre à votre fonction.
- `namespace ExampleLambda` : définit l'espace de noms. En C#, il n'est pas nécessaire que le nom de l'espace de noms corresponde au nom du fichier.
- `public class Order {...}` : définit la forme de l'événement d'entrée attendu.
- `public class OrderHandler {...}` : définit votre classe C#. Vous y définirez la méthode du gestionnaire principal ainsi que toutes autres méthodes auxiliaires.
- `private static readonly AmazonS3Client s3Client = new();` : initialise un client Amazon S3 avec la chaîne de fournisseur d'informations d'identification par défaut, en dehors de la méthode du gestionnaire principal. Cela oblige Lambda à exécuter ce code pendant la phase d'initialisation.
- `public async ... HandleRequest (Order order, ILambdaContext context)` : il s'agit de la méthode de gestion principale, qui contient la logique principale de votre application.
- `private async Task UploadReceiptToS3(...) {}` : il s'agit d'une méthode auxiliaire référencée par la méthode de gestion principale `handleRequest`.

Comme cette fonction nécessite un client SDK Amazon S3, vous devez l'ajouter aux dépendances de votre projet. Vous pouvez également le faire en accédant à `src/ExampleCS` et en exécutant la commande suivante :

```
dotnet add package AWSSDK.S3
```

Ajouter des informations de métadonnées au aws-lambda-tools-defaults fichier .json

Par défaut, le fichier `aws-lambda-tools-defaults.json` généré ne contient aucune information `profile` ou `region` relative à votre fonction. Mettez également à jour la chaîne `function-handler` avec la bonne valeur (`ExampleCS::ExampleLambda.OrderHandler::HandleRequest`). Vous pouvez effectuer cette mise à jour manuellement et ajouter les métadonnées nécessaires pour utiliser un profil d'identification et une région spécifiques pour votre fonction. Par exemple, votre fichier `aws-lambda-tools-defaults.json` doit ressembler à ceci :

```
{
  "Information": [
    "This file provides default values for the deployment wizard inside Visual Studio
 and the AWS Lambda commands added to the .NET Core CLI.",
    "To learn more about the Lambda commands with the .NET Core CLI execute the
 following command at the command line in the project root directory.",
    "dotnet lambda help",
    "All the command line options for the Lambda command can be specified in this
 file."
  ],
  "profile": "default",
  "region": "us-east-1",
  "configuration": "Release",
  "function-architecture": "x86_64",
  "function-runtime": "dotnet8",
  "function-memory-size": 512,
  "function-timeout": 30,
  "function-handler": "ExampleCS::ExampleLambda.OrderHandler::HandleRequest"
}
```

Pour que cette fonction fonctionne correctement, son rôle d'exécution doit autoriser l'action `s3:PutObject`. Assurez-vous également de définir la variable d'environnement `RECEIPT_BUCKET`. Après une invocation réussie, le compartiment Amazon S3 doit contenir un fichier de reçu.

Gestionnaires de bibliothèques de classes

L'exemple de code principal de cette page illustre un gestionnaire de bibliothèques de classes. La structure des gestionnaires de bibliothèques de classes est la suivante :

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace NAMESPACE;

...

public class CLASSNAME {
    public async Task<string> METHODNAME (...) {
    ...
    }
}
```

Lorsque vous créez une fonction Lambda, vous devez fournir à Lambda des informations sur le gestionnaire de votre fonction sous la forme d'une chaîne dans le champ Gestionnaire. Cette chaîne indique à Lambda quelle méthode de votre code doit être exécutée lorsque votre fonction est invoquée. En C#, pour les gestionnaires de bibliothèques de classes, le format de la chaîne du gestionnaire est `ASSEMBLY::TYPE::METHOD`, où :

- `ASSEMBLY` est le nom du fichier d'assemblage .NET de votre application. Si vous utilisez la CLI `Amazon.Lambda.Tools` pour créer votre application et que vous ne définissez pas le nom de l'assemblage à l'aide de la propriété `AssemblyName` dans le fichier `.csproj`, alors `ASSEMBLY` est simplement le nom de votre fichier `.csproj`.
- `TYPE` est le nom complet du type de gestionnaire, à savoir `NAMESPACE.CLASSNAME`.
- `METHOD` est le nom de la méthode de gestionnaire principale de fonction dans votre code, à savoir `METHODNAME`.

Dans l'exemple de code principal illustré sur cette page, si l'assemblage est nommé `ExampleCS`, alors la chaîne du gestionnaire complète est `ExampleCS::ExampleLambda.OrderHandler::HandleRequest`.

Gestionnaires d'assemblages exécutables

Vous pouvez également définir des fonctions Lambda en C# sous forme d'assemblage exécutable. Les gestionnaires d'assemblages exécutables utilisent la fonctionnalité d'instructions de haut niveau de C#, dans laquelle le compilateur génère la méthode `Main()` et y insère le code de votre fonction. Lors de l'utilisation d'assemblages exécutables, l'exécution Lambda doit être amorcée. Pour ce faire, utilisez la méthode `LambdaBootstrapBuilder.Create` dans votre code. Les entrées de cette méthode sont la fonction principale du gestionnaire ainsi que le sérialiseur Lambda à utiliser. Voici un exemple de gestionnaire d'assemblages exécutables en C# :

```
namespace GetProductHandler;

IDatabaseRepository repo = new DatabaseRepository();

await LambdaBootstrapBuilder.Create<APIGatewayProxyRequest>(Handler, new
 DefaultLambdaJsonSerializer())
    .Build()
    .RunAsync();

async Task<APIGatewayProxyResponse> Handler(APIGatewayProxyRequest apigProxyEvent,
 ILambdaContext context)
```

```
{
    var id = apigProxyEvent.PathParameters["id"];
    var databaseRecord = await this.repo.GetById(id);

    return new APIGatewayProxyResponse
    {
        StatusCode = (int)HttpStatusCode.OK,
        Body = JsonSerializer.Serialize(databaseRecord)
    };
};
```

Dans le champ Gestionnaire pour les gestionnaires d'assemblages exécutables, la chaîne du gestionnaire qui indique à Lambda comment exécuter votre code est le nom de l'assemblage. Dans cet exemple, il s'agit de `GetProductHandler`.

Signatures de gestionnaire valides pour les fonctions C#

En C#, les signatures de gestionnaire Lambda valides prennent entre 0 et 2 arguments. Généralement, la signature de votre gestionnaire comporte deux arguments, comme indiqué dans l'exemple principal :

```
public async Task<string> HandleRequest(Order order, ILambdaContext context)
```

Lorsque vous fournissez deux arguments, le premier doit être l'entrée de l'événement et le second l'objet de contexte Lambda. Les deux arguments sont facultatifs. Par exemple, les signatures de gestionnaire Lambda suivantes sont également valides en C# :

- `public async Task<string> HandleRequest()`
- `public async Task<string> HandleRequest(Order order)`
- `public async Task<string> HandleRequest(ILambdaContext context)`

Outre la syntaxe de base de la signature du gestionnaire, il existe un certain nombre de restrictions supplémentaires :

- Vous ne pouvez pas utiliser le mot clé `unsafe` dans la signature du gestionnaire. Cependant, vous pouvez utiliser le contexte `unsafe` dans la méthode du gestionnaire et ses dépendances. Pour de plus amples informations, veuillez consulterdangereux (référence C#)sur le site Web de documentation de Microsoft.

- Le gestionnaire ne doit pas utiliser le mot clé `params`, ni utiliser `ArgIterator` comme paramètre d'entrée ou de retour. Ces mots clés prennent en charge un nombre variable de paramètres. Le nombre maximum d'arguments que votre gestionnaire peut accepter est de deux.
- Le gestionnaire ne peut pas être une méthode générique. En d'autres termes, il ne peut pas utiliser de paramètres de type générique, comme <T>.
- Lambda ne prend pas en charge les gestionnaires asynchrones contenant `async void` dans la signature.

Convention de nommage du gestionnaire

Les gestionnaires Lambda en C# ne sont pas soumis à des restrictions de dénomination strictes. Toutefois, vous devez vous assurer que vous fournissez la bonne chaîne de gestionnaire à Lambda lorsque vous déployez votre fonction. La bonne chaîne de gestionnaire varie selon que vous déployez un gestionnaire de bibliothèques de classes ou un gestionnaire d'assemblages exécutables.

Bien que vous puissiez utiliser n'importe quel nom pour votre gestionnaire, les noms de fonctions en C# sont généralement disponibles. PascalCase De plus, même si le nom du fichier ne doit pas nécessairement correspondre au nom de la classe ou au nom du gestionnaire, il est généralement recommandé d'utiliser un nom de fichier comme `OrderHandler.cs` si le nom de votre classe est `OrderHandler`. Dans cet exemple, vous pouvez modifier le nom de fichier de `Function.cs` à `OrderHandler.cs`.

Sérialisation dans les fonctions Lambda C#

JSON est le format d'entrée le plus courant et standard pour les fonctions Lambda. Dans cet exemple, la fonction exige une entrée similaire à l'exemple suivant :

```
{
    "orderId": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

En C#, vous pouvez définir la forme de l'événement d'entrée attendu dans une classe. Dans cet exemple, nous définissons la classe `Order` pour modéliser cette entrée :

```
public class Order
```

```
{
    public string OrderId { get; set; } = string.Empty;
    public double Amount { get; set; }
    public string Item { get; set; } = string.Empty;
}
```

Si votre fonction Lambda utilise des types d'entrée ou de sortie autres qu'un objet `Stream`, vous devez ajouter une bibliothèque de sérialisation à votre application. Cela vous permet de convertir l'entrée JSON en une instance de la classe que vous avez définie. Il existe deux méthodes de sérialisation pour les fonctions C# dans Lambda : la sérialisation basée sur la réflexion et la sérialisation générée par la source.

Sérialisation basée sur la réflexion

AWS fournit des bibliothèques prédéfinies que vous pouvez rapidement ajouter à votre application. Ces bibliothèques implémentent la sérialisation à l'aide de la réflexion. Utilisez l'un des packages suivants pour implémenter la sérialisation basée sur la réflexion :

- `Amazon.Lambda.Serialization.SystemTextJson` : dans le backend, ce package utilise `System.Text.Json` pour effectuer des tâches de sérialisation.
- `Amazon.Lambda.Serialization.Json` : dans le backend, ce package utilise `Newtonsoft.Json` pour effectuer des tâches de sérialisation.

Vous pouvez également créer votre propre bibliothèque de sérialisation en implémentant l'interface `ILambdaSerializer`, disponible dans la bibliothèque `Amazon.Lambda.Core`. Cette interface définit deux méthodes :

- `T Deserialize<T>(Stream requestStream);`

  Vous implémentez cette méthode afin de désérialiser la charge utile de la demande à partir de l'API `Invoke` dans l'objet qui est transféré à votre gestionnaire de fonction Lambda.
- `T Serialize<T>(T response, Stream responseStream);`

  Vous implémentez cette méthode pour sérialiser le résultat renvoyé à partir de votre gestionnaire de fonction Lambda dans la charge utile de la réponse renvoyée par l'opération d'API `Invoke`.

L'exemple principal de cette page utilise la sérialisation basée sur la réflexion. La sérialisation basée sur la réflexion fonctionne immédiatement AWS Lambda et ne nécessite aucune configuration

supplémentaire, ce qui en fait un bon choix en termes de simplicité. Toutefois, elle nécessite une utilisation plus importante de la mémoire de la fonction. Vous pourriez également constater des latences de fonction plus élevées en raison de la réflexion dans l'environnement d'exécution.

Sérialisation générée par la source

Avec la sérialisation générée par la source, le code de sérialisation est généré au moment de la compilation. Cela supprime le besoin de réflexion et peut améliorer les performances de votre fonction. Pour utiliser la sérialisation générée par la source dans votre fonction, vous devez procéder comme suit :

- Créez une nouvelle classe partielle qui hérite de `JsonSerializerContext`, en ajoutant des attributs `JsonSerializable` pour tous les types qui nécessitent une sérialisation ou une désérialisation.
- Configurez le `LambdaSerializer` afin d'utiliser un `SourceGeneratorLambdaJsonSerializer<T>`.
- Mettez à jour toute sérialisation ou désérialisation manuelle dans le code de votre application afin d'utiliser la classe nouvellement créée.

L'exemple suivant montre comment vous pouvez modifier l'exemple principal de cette page, qui utilise la sérialisation basée sur la réflexion, pour utiliser plutôt la sérialisation générée par la source.

```
using System.Text.Json;
using System.Text.Json.Serialization;

...

public class Order
{
    public string OrderId { get; set; } = string.Empty;
    public double Amount { get; set; }
    public string Item { get; set; } = string.Empty;
}

[JsonSerializable(typeof(Order))]
public partial class OrderJsonContext : JsonSerializerContext {}

public class OrderHandler
{
```

```
    ...

    public async Task<string> HandleRequest(string input, ILambdaContext context)
    {

    var order = JsonSerializer.Deserialize(input, OrderJsonContext.Default.Order);

    ...

    }

}
```

La sérialisation générée par la source nécessite davantage de configuration que la sérialisation basée sur la réflexion. Cependant, les fonctions utilisant la sérialisation générée par la source ont tendance à utiliser moins de mémoire et à offrir de meilleures performances en raison de la génération du code au moment de la compilation. Pour éliminer les démarrages à froid des fonctions, envisagez de passer à la sérialisation générée par la source.

> **Note**
>
> Si vous souhaitez utiliser la ahead-of-time compilation native (AOT) avec Lambda, vous devez utiliser la sérialisation générée par la source.

Fonctions basées sur des fichiers

Introduites dans .NET 10, les applications basées sur des fichiers vous permettent de créer des applications .NET à partir d'un seul `.cs` fichier, sans structure de `.csproj` fichier ou de répertoire. Lambda prend en charge les fonctions basées sur des fichiers, à commencer par .NET 10. Ils offrent un moyen simple et léger de créer des fonctions Lambda en C#.

Le moyen le plus rapide de commencer à créer une fonction Lambda basée sur un fichier C# consiste à utiliser le package. `Amazon.Lambda.Templates` Pour installer le paquet, exécutez la commande suivante :

```
dotnet new install Amazon.Lambda.Templates
```

Ensuite, créez un exemple de fonction Lambda basée sur un fichier C# :

```
dotnet new lambda.FileBased -n MyLambdaFunction
```

Les fonctions basées sur des fichiers utilisent des gestionnaires d'assemblages exécutables. Vous devez donc inclure le `Amazon.Lambda.RuntimeSupport` NuGet package et utiliser la `LambdaBootstrapBuilder.Create` méthode pour enregistrer la fonction de gestionnaire .NET pour le type d'événement et démarrer le client d'exécution .NET Lambda.

Les fonctions basées sur des fichiers utilisent .NET Native AOT par défaut, ce qui nécessite une sérialisation générée par la source. Vous pouvez désactiver l'AOT natif en le spécifiant `#:property PublishAot=false` dans votre fichier source. Pour plus d'informations sur l'utilisation de l'AOT natif dans Lambda, consultez. the section called “Compilation anticipée native”

Accès et utilisation de l’objet de contexte Lambda

L’objet de contexte Lambda contient des informations sur l’invocation, la fonction et l’environnement d’exécution. Dans cet exemple, l’objet de contexte est de type `Amazon.Lambda.Core.ILambdaContext`, et constitue le deuxième argument de la fonction de gestion principale.

```
public async Task<string> HandleRequest(Order order, ILambdaContext context) {
    ...
}
```

L’objet de contexte est une entrée facultative. Pour plus d’informations sur les signatures de gestionnaire acceptées valides, consultez the section called “Signatures de gestionnaire valides pour les fonctions C#”.

L'objet de contexte est utile pour générer des journaux de fonctions pour Amazon CloudWatch. Vous pouvez utiliser la méthode `context.getLogger()` pour obtenir un objet `LambdaLogger` pour l’enregistrement. Dans cet exemple, nous pouvons utiliser l’enregistreur pour enregistrer un message d’erreur si le traitement échoue pour une raison quelconque :

```
context.Logger.LogError($"Failed to process order: {ex.Message}");
```

En dehors de la journalisation, vous pouvez également utiliser l’objet de contexte pour surveiller les fonctions. Pour plus d’informations sur la copie d’objets, consultez the section called “Contexte”.

Utilisation de la SDK pour .NET v3 dans votre gestionnaire

Vous utiliserez souvent les fonctions Lambda pour interagir avec d'autres AWS ressources ou pour les mettre à jour. Le moyen le plus simple d'interagir avec ces ressources est d'utiliser la SDK pour .NET version 3.

> **Note**
>
> Le SDK pour .NET (v2) est obsolète. Nous vous recommandons de n'utiliser que la SDK pour .NET version 3.

Vous pouvez ajouter des dépendances du SDK à votre projet à l'aide de la commande `Amazon.Lambda.Tools` suivante :

```
dotnet add package <package_name>
```

Par exemple, dans l'exemple principal sur cette page, nous devons utiliser l'API Amazon S3 pour charger un reçu sur S3. Nous pouvons importer le client du SDK Amazon S3 à l'aide de la commande suivante :

```
dotnet add package AWSSDK.S3
```

Cette commande ajoute la dépendance à votre projet. Vous devriez également voir une ligne similaire à la suivante dans le fichier `.csproj` de votre projet :

```
<PackageReference Include="AWSSDK.S3" Version="3.7.2.18" />
```

Importez ensuite les dépendances directement dans votre code C# :

```
using Amazon.S3;
using Amazon.S3.Model;
```

L'exemple de code initialise ensuite un client Amazon S3 (en utilisant la chaîne de fournisseur d'informations d'identification par défaut) comme suit :

```
private static readonly AmazonS3Client s3Client = new();
```

Dans cet exemple, nous avons initialisé notre client Amazon S3 en dehors de la fonction de gestionnaire principale pour éviter d'avoir à l'initialiser à chaque fois que nous invoquons notre fonction. Après avoir initialisé votre client SDK, vous pouvez l'utiliser pour interagir avec d'autres AWS services. L'exemple de code appelle l'API Amazon S3 `PutObject` comme suit :

```
var putRequest = new PutObjectRequest
{
    BucketName = bucketName,
    Key = key,
    ContentBody = receiptContent,
    ContentType = "text/plain"
};

await s3Client.PutObjectAsync(putRequest);
```

Accès aux variables d'environnement

Dans le code de votre gestionnaire, vous pouvez référencer n'importe quelle variable d'environnement à l'aide de la méthode `System.Environment.GetEnvironmentVariable`. Dans cet exemple, nous référençons la variable d'environnement `RECEIPT_BUCKET` définie à l'aide des lignes de code suivantes :

```
string? bucketName = Environment.GetEnvironmentVariable("RECEIPT_BUCKET");
if (string.IsNullOrWhiteSpace(bucketName))
{
    throw new ArgumentException("RECEIPT_BUCKET environment variable is not set");
}
```

Utilisation de l'état global

Lambda exécute votre code statique et le constructeur de classe pendant la phase d'initialisation avant d'invoquer votre fonction pour la première fois. Les ressources créées pendant l'initialisation restent en mémoire entre les invocations, ce qui vous évite d'avoir à les créer chaque fois que vous invoquez votre fonction.

Dans l'exemple de code, le code d'initialisation du client S3 se trouve en dehors de la méthode du gestionnaire principal. L'exécution initialise le client avant que la fonction exécute son premier événement, ce qui peut entraîner des délais de traitement plus longs. Les événements suivants sont beaucoup plus rapides, car Lambda n'a pas besoin d'initialiser à nouveau le client.

Simplifiez le code de la fonction à l'aide du cadre d'annotations Lambda

Les annotations Lambda constituent un cadre pour .NET 8 qui simplifie l'écriture des fonctions Lambda en utilisant C#. Le cadre d'annotations utilise des générateurs de source pour générer un code qui traduit le modèle de programmation Lambda en code simplifié. Grâce au cadre d'annotations, vous pouvez remplacer une grande partie du code d'une fonction Lambda écrite à l'aide du modèle de programmation habituel. Le code écrit à l'aide du cadre utilise des expressions plus simples qui vous permettent de vous concentrer sur votre logique commerciale. Consultez Amazon.Lambda.Annotations dans la documentation NuGet pour des exemples.

Pour un exemple d'application complète utilisant les annotations Lambda, consultez l'PhotoAssetManagerexemple dans le référentiel. `awsdocs/aws-doc-sdk-examples` GitHub Le fichier `Function.cs` principal dans le répertoire `PamApiAnnotations` utilise des annotations Lambda. À titre de comparaison, le répertoire `PamApi` contient des fichiers équivalents écrits à l'aide du modèle de programmation Lambda habituel.

Injection de dépendances grâce au cadre d'annotations Lambda

Vous pouvez également utiliser le cadre d'annotations Lambda pour ajouter l'injection de dépendance à vos fonctions Lambda en utilisant une syntaxe que vous connaissez bien. Lorsque vous ajoutez un attribut `[LambdaStartup]` à un fichier `Startup.cs`, le cadre d'annotations Lambda génère le code nécessaire au moment de la compilation.

```
[LambdaStartup]
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSingleton<IDatabaseRepository, DatabaseRepository>();
    }
}
```

Votre fonction Lambda peut injecter des services en utilisant l'injection de constructeur ou en injectant dans des méthodes individuelles à l'aide de l'attribut `[FromServices]`.

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;
```

```
public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function(IDatabaseRepository repo)
    {
        this._repo = repo;
    }

    [LambdaFunction]
    [HttpApi(LambdaHttpMethod.Get, "/product/{id}")]
    public async Task<Product> FunctionHandler([FromServices] IDatabaseRepository
 repository, string id)
    {
        return await this._repo.GetById(id);
    }
}
```

Pratiques exemplaires de codage pour les fonctions Lambda C#

Respectez les directives de la liste suivante pour utiliser les pratiques exemplaires de codage lors de la création de vos fonctions Lambda :

- Séparez le gestionnaire Lambda de votre logique principale. Cela vous permet de créer une fonction testable plus unitaire.
- Contrôlez les dépendances du package de déploiement de vos fonctions. L'environnement d'exécution AWS Lambda contient un certain nombre de bibliothèques. Pour activer le dernier ensemble de mises à jour des fonctionnalités et de la sécurité, Lambda met régulièrement à jour ces bibliothèques. Ces mises à jour peuvent introduire de subtiles modifications dans le comportement de votre fonction Lambda. Pour disposer du contrôle total des dépendances que votre fonction utilise, empaquetez toutes vos dépendances avec votre package de déploiement.
- Réduisez la complexité de vos dépendances. Privilégiez les infrastructures plus simples qui se chargent rapidement au démarrage de l'environnement d'exécution.
- Réduisez la taille de votre package de déploiement selon ses besoins d'exécution. Cela contribue à réduire le temps nécessaire au téléchargement et à la décompression de votre package de déploiement avant l'invocation. Pour les fonctions créées dans .NET, évitez de télécharger l'intégralité de la bibliothèque du AWS SDK dans le cadre de votre package de déploiement. À la place, appuyez-vous de façon sélective sur les modules qui sélectionnent les composants du

kit SDK dont vous avez besoin (par exemple, DynamoDB, modules du kit SDK Amazon S3 et bibliothèques principales Lambda).

Tirez parti de la réutilisation de l'environnement d'exécution pour améliorer les performances de votre fonction. Initialisez les clients SDK et les connexions à la base de données en dehors du gestionnaire de fonctions et mettez en cache les actifs statiques localement dans le répertoire `/tmp`. Les invocations ultérieures traitées par la même instance de votre fonction peuvent réutiliser ces ressources. Cela permet d'économiser des coûts, tout en réduisant le temps d'exécution de la fonction.

Pour éviter des éventuelles fuites de données entre les invocations, n'utilisez pas l'environnement d'exécution pour stocker des données utilisateur, des événements ou d'autres informations ayant un impact sur la sécurité. Si votre fonction repose sur un état réversible qui ne peut pas être stocké en mémoire dans le gestionnaire, envisagez de créer une fonction distincte ou des versions distinctes d'une fonction pour chaque utilisateur.

Utilisez une directive keep-alive pour maintenir les connexions persistantes. Lambda purge les connexions inactives au fil du temps. Si vous tentez de réutiliser une connexion inactive lorsque vous invoquez une fonction, cela entraîne une erreur de connexion. Pour maintenir votre connexion persistante, utilisez la directive Keep-alive associée à votre environnement d'exécution. Pour obtenir un exemple, consultez Réutilisation des connexions avec Keep-Alive dans Node.js.

Utilisez des variables d'environnement pour transmettre des paramètres opérationnels à votre fonction. Par exemple, si vous écrivez dans un compartiment Amazon S3 au lieu de coder en dur le nom du compartiment dans lequel vous écrivez, configurez le nom du compartiment comme variable d'environnement.

Évitez d'utiliser des invocations récursives dans votre fonction Lambda, lorsque la fonction s'invoque elle-même ou démarre un processus susceptible de l'invoquer à nouveau. Cela peut entraîner un volume involontaire d'invocations de fonction et des coûts accrus. Si vous constatez un volume involontaire d'invocations, définissez immédiatement la simultanéité réservée à la fonction sur `0` afin de limiter toutes les invocations de la fonction, pendant que vous mettez à jour le code.

N'utilisez pas de code non documenté ni public APIs dans votre code de fonction Lambda. Pour les AWS Lambda environnements d'exécution gérés, Lambda applique régulièrement des mises à jour de sécurité et fonctionnelles aux applications internes de Lambda. APIs Ces mises à jour internes de l'API peuvent être rétroincompatibles, ce qui peut entraîner des conséquences imprévues, telles que

des échecs d'invocation si votre fonction dépend de ces mises à jour non publiques. APIs Consultez la référence de l'API pour obtenir une liste des API accessibles au public APIs.

Écriture du code idempotent. L'écriture de code idempotent pour vos fonctions garantit ne gestion identique des événements dupliqués. Votre code doit valider correctement les événements et gérer correctement les événements dupliqués. Pour de plus amples informations, veuillez consulterComment faire en sorte que ma fonction Lambda soit idempotente ?.

Créez et déployez des fonctions Lambda C# à l'aide des archives de fichiers .zip

Un package de déploiement .NET (archive de fichier .zip) contient l'assemblage compilé de votre fonction ainsi que toutes les dépendances de l'assemblage. Le package contient également un fichier *proj*`.deps.json`. Il indique à l'exécution .NET toutes les dépendances de votre fonction et un fichier *proj*`.runtimeconfig.json`, qui est utilisé pour configurer l'exécution.

Pour déployer des fonctions Lambda individuelles, vous pouvez utiliser la CLI .NET Lambda Global de `Amazon.Lambda.Tools`. L'utilisation de la commande `dotnet lambda deploy-function` crée automatiquement un package de déploiement .zip et le déploie sur Lambda. Cependant, nous vous recommandons d'utiliser des cadres tels que le AWS Serverless Application Model (AWS SAM) ou le AWS Cloud Development Kit (AWS CDK) pour déployer vos applications .NET sur AWS.

Les applications sans serveur comprennent généralement une combinaison de fonctions Lambda et d'autres Services AWS gérés qui fonctionnent ensemble pour exécuter une tâche commerciale particulière. AWS SAM et AWS CDK simplifient la création et le déploiement de fonctions Lambda avec d'autres Services AWS à grande échelle. La spécification du modèle AWS SAM fournit une syntaxe simple et propre pour décrire les fonctions Lambda, les API, les autorisations, les configurations et les autres ressources AWS qui composent votre application sans serveur. Grâce au AWS CDK, vous pouvez définir l'infrastructure cloud en tant que code pour vous aider à créer des applications fiables, évolutives et rentables dans le cloud à l'aide de langages de programmation et de cadres modernes tels que .NET. Le AWS CDK et le AWS SAM utilisent tous deux la CLI .NET Lambda Global pour empaqueter vos fonctions.

Bien qu'il soit possible d'utiliser des couches Lambda avec des fonctions en C# à l'aide de la CLI .NET Core, nous vous le déconseillons. Les fonctions en C# qui utilisent des couches chargent manuellement les assemblages partagés dans la mémoire pendant le Phase d'initialisation, ce qui peut augmenter les temps de démarrage à froid. Incluez plutôt tout le code partagé au moment de la compilation pour éviter d'affecter les performances du chargement des assemblages au moment de l'exécution.

Vous trouverez des instructions pour créer et déployer des fonctions Lambda .NET en utilisant le AWS SAM, le AWS CDK et la CLI .NET Lambda Global dans les sections suivantes.

Rubriques

- Utilisation de la CLI .NET Lambda Global
- Déploiement de fonctions Lambda C# à l'aide d'AWS SAM

- Déploiement de fonctions Lambda C# à l'aide d'AWS CDK
- Déployez des applications ASP.NET

Utilisation de la CLI .NET Lambda Global

La CLI .NET et l'extension .NET Lambda Global Tools (`Amazon.Lambda.Tools`) offrent un moyen multiplateforme de créer des applications Lambda basées sur .NET, de les empaqueter et de les déployer dans Lambda. Dans cette section, vous apprendrez à créer de nouveaux projets Lambda .NET à l'aide de la CLI .NET et des modèles Amazon Lambda, et à les empaqueter et les déployer en utilisant `Amazon.Lambda.Tools`

Rubriques

- Prérequis
- Création de projets .NET à l'aide de la CLI .NET
- Déploiement de projets .NET à l'aide de la CLI .NET
- Utilisation de couches Lambda avec l'interface de ligne de commande .NET

Prérequis

Kit SDK .NET 8

Si vous ne l'avez pas encore fait, installez le kit SDK .NET 8 et l'environnement d'exécution.

Modèles de projet .NET d'Amazon.Lambda.Templates AWS

Pour générer votre code de fonction Lambda, utilisez le package NuGet Amazon.Lambda.Templates. Pour installer ce package de modèle, exécutez la commande suivante :

```
dotnet new install Amazon.Lambda.Templates
```

Outils globaux CLI .NET d'Amazon.Lambda.Tools AWS

Pour créer vos fonctions Lambda, utilisez l'Amazon.Lambda.Tools extension outils globaux .NET. Pour installer Amazon.Lambda.Tools, exécutez la commande suivante :

```
dotnet tool install -g Amazon.Lambda.Tools
```

Pour en savoir plus sur l'extension CLI .NET Amazon.Lambda.Tools, consultez le référentiel Extensions pour la CLI .NET AWS dans GitHub.

Création de projets .NET à l'aide de la CLI .NET

Dans la CLI .NET, utilisez la commande `dotnet new` pour créer des projets .NET à partir de la ligne de commande. Lambda propose des modèles supplémentaires à l'aide du package NuGet `Amazon.Lambda.Templates`.

Après avoir installé ce package, exécutez la commande suivante pour obtenir la liste des modèles disponibles.

```
dotnet new list
```

Pour examiner les détails relatifs à un modèle, utilisez l'option `help`. Par exemple, pour obtenir des détails sur le modèle `lambda.EmptyFunction`, exécutez la commande suivante.

```
dotnet new lambda.EmptyFunction --help
```

Pour créer un modèle de base pour une fonction Lambda .NET, utilisez le modèle `lambda.EmptyFunction`. Celui-ci crée une fonction simple qui prend une chaîne comme entrée et la convertit en majuscules à l'aide de la méthode `ToUpper`. Ce modèle prend en charge les options suivantes :

- `--name` – Nom de la fonction.
- `--region` – Région AWS dans laquelle la fonction doit être créée.
- `--profile` – Nom d'un profil dans votre fichier d'informations d'identification AWS SDK pour .NET. Pour en savoir plus sur les profils d'informations d'identification dans .NET, consultez la section Configurer les informations d'identification AWS dans le Guide du développeur du kit SDK AWS pour .NET.

Dans cet exemple, nous créons une nouvelle fonction vide nommée `myDotnetFunction` en utilisant le profil et les paramètres par défaut Région AWS :

```
dotnet new lambda.EmptyFunction --name myDotnetFunction
```

Cette commande crée les fichiers et répertoires suivants dans le répertoire de votre projet.

```
### myDotnetFunction
    ### src
    #   ### myDotnetFunction
    #       ### Function.cs
    #       ### Readme.md
    #       ### aws-lambda-tools-defaults.json
    #       ### myDotnetFunction.csproj
    ### test
        ### myDotnetFunction.Tests
            ### FunctionTest.cs
            ### myDotnetFunction.Tests.csproj
```

Sous le répertoire `src/myDotnetFunction`, examinez les fichiers suivants :

- aws-lambda-tools-defaults.json : c'est ici que vous spécifiez les options de ligne de commande lors du déploiement de votre fonction Lambda. Exemples :

  ```
    "profile" : "default",
    "region" : "us-east-2",
    "configuration" : "Release",
    "function-architecture": "x86_64",
    "function-runtime":"dotnet8",
    "function-memory-size" : 256,
    "function-timeout" : 30,
    "function-handler" : "myDotnetFunction::myDotnetFunction.Function::FunctionHandler"
  ```

- Function.cs : code de fonction de votre gestionnaire Lambda. Il s'agit d'un modèle C # qui inclut la bibliothèque `Amazon.Lambda.Core` par défaut et un attribut `LambdaSerializer` par défaut. Pour plus d'informations sur les conditions de sérialisation et les options, consultez Sérialisation dans les fonctions Lambda C#. Il inclut également un exemple de fonction que vous pouvez modifier pour appliquer votre code de fonction Lambda.

  ```
  using Amazon.Lambda.Core;

  // Assembly attribute to enable the Lambda function's JSON input to be converted into
   a .NET class.
  [assembly:
   LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializ

  namespace myDotnetFunction;
  ```

```
public class Function
{

    /// <summary>
    /// A simple function that takes a string and does a ToUpper
    /// </summary#
    /// <param name="input"></param>
    /// <param name="context"></param>
    /// <returns></returns>
    public string FunctionHandler(string input, ILambdaContext context)
    {
        return input.ToUpper();
    }
}
```

- myDotnetFunction.csproj : un fichier MSBuild qui répertorie les fichiers et les assemblages qui composent votre application.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <GenerateRuntimeConfigurationFiles>true</GenerateRuntimeConfigurationFiles>
    <AWSProjectType>Lambda</AWSProjectType>
    <!-- This property makes the build directory similar to a publish directory and
 helps the AWS .NET Lambda Mock Test Tool find project dependencies. -->
    <CopyLocalLockFileAssemblies>true</CopyLocalLockFileAssemblies>
    <!-- Generate ready to run images during publishing to improve cold start time.
 -->
    <PublishReadyToRun>true</PublishReadyToRun>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Amazon.Lambda.Core" Version="2.2.0" />
    <PackageReference Include="Amazon.Lambda.Serialization.SystemTextJson"
 Version="2.4.0" />
  </ItemGroup>
</Project>
```

- Readme : utilisez ce fichier pour documenter votre fonction Lambda.

Sous le répertoire `myfunction/test`, examinez les fichiers suivants :

- myDotnetFunction.Tests.csproj : comme indiqué précédemment, il s'agit d'un fichier MSBuild qui répertorie les fichiers et les assemblages qui composent votre projet de test. Notez également qu'il comprend aussi la bibliothèque `Amazon.Lambda.Core`, ce qui vous permet d'intégrer de manière transparente tout modèle Lambda requis pour tester votre fonction.

```
<Project Sdk="Microsoft.NET.Sdk">
   ...

    <PackageReference Include="Amazon.Lambda.Core" Version="2.2.0 " />
   ...
```

- FunctionTest.cs : le même fichier de modèle de code C# que celui inclus dans le répertoire `src`. Modifiez ce fichier pour mettre en miroir le code de production de votre fonction, et testez-le avant de télécharger votre fonction Lambda sur un environnement de production.

```
using Xunit;
using Amazon.Lambda.Core;
using Amazon.Lambda.TestUtilities;

using MyFunction;

namespace MyFunction.Tests
{
    public class FunctionTest
    {
        [Fact]
        public void TestToUpperFunction()
        {

            // Invoke the lambda function and confirm the string was upper cased.
            var function = new Function();
            var context = new TestLambdaContext();
            var upperCase = function.FunctionHandler("hello world", context);

            Assert.Equal("HELLO WORLD", upperCase);
        }
    }
}
```

Déploiement de projets .NET à l'aide de la CLI .NET

Pour créer votre package de déploiement et le déployer sur Lambda, utilisez les outils CLI `Amazon.Lambda.Tools`. Pour déployer votre fonction à partir des fichiers que vous avez créés dans les étapes précédentes, accédez d'abord au dossier contenant le fichier `.csproj` de votre fonction.

```
cd myDotnetFunction/src/myDotnetFunction
```

Pour déployer votre code sur Lambda sous la forme d'un package de déploiement .zip, exécutez la commande suivante. Choisissez le nom de votre fonction.

```
dotnet lambda deploy-function myDotnetFunction
```

Pendant le déploiement, l'assistant vous invite à sélectionner un the section called "Rôle d'exécution (autorisations pour les fonctions d'accéder à d'autres ressources)". Pour cet exemple, sélectionnez le `lambda_basic_role`.

Une fois votre fonction déployée, vous pouvez la tester dans le cloud en utilisant la commande `dotnet lambda invoke-function`. Pour l'exemple de code dans le modèle `lambda.EmptyFunction`, vous pouvez tester votre fonction en transmettant une chaîne en utilisant l'option `--payload`.

```
dotnet lambda invoke-function myDotnetFunction --payload "Just checking if everything
 is OK"
```

Si votre fonction a été déployée avec succès, vous devriez obtenir une sortie similaire à celle qui suit.

```
dotnet lambda invoke-function myDotnetFunction --payload "Just checking if everything
 is OK"
Amazon Lambda Tools for .NET Core applications (5.8.0)
Project Home: https://github.com/aws/aws-extensions-for-dotnet-cli, https://github.com/
aws/aws-lambda-dotnet

Payload:
"JUST CHECKING IF EVERYTHING IS OK"

Log Tail:
START RequestId: id Version: $LATEST
END RequestId: id
```

```
REPORT RequestId: id  Duration: 0.99 ms       Billed Duration: 1 ms         Memory
 Size: 256 MB     Max Memory Used: 12 MB
```

Utilisation de couches Lambda avec l'interface de ligne de commande .NET

> **Note**
>
> Bien qu'il soit possible d'utiliser des couches avec dans .NET, nous vous le déconseillons. Les fonctions en .NET qui utilisent des couches chargent manuellement les assemblages partagés dans la mémoire pendant la phase `Init`, ce qui peut augmenter les temps de démarrage à froid. Incluez plutôt tout le code partagé au moment de la compilation pour tirer parti des optimisations intégrées du compilateur .NET.

L'interface de ligne de commande .NET prend en charge les commandes qui vous aident à publier des couches et à déployer des fonctions C# consommant des couches. Pour publier une couche dans un compartiment Amazon S3 spécifié, exécutez la commande suivante dans le même répertoire que votre fichier `.csproj` :

```
dotnet lambda publish-layer <layer_name> --layer-type runtime-package-store --s3-
bucket <s3_bucket_name>
```

Ensuite, lorsque vous déployez votre fonction à l'aide de l'interface de ligne de commande .NET, spécifiez l'ARN de couche à consommer dans la commande suivante :

```
dotnet lambda deploy-function <function_name> --function-layers arn:aws:lambda:us-
east-1:123456789012:layer:layer-name:1
```

Pour un exemple complet de fonction Hello World, consultez l'exemple blank-csharp-with-layer.

Déploiement de fonctions Lambda C# à l'aide d'AWS SAM

L'AWS Serverless Application Model (AWS SAM) est une boîte à outils qui permet de rationaliser le processus de création et d'exécution d'applications sans serveur sur AWS. Vous définissez les ressources de votre application dans un modèle YAML ou JSON et vous utilisez l'interface de ligne de commande AWS SAM (AWS SAM CLI) pour créer, empaqueter et déployer vos applications. Lorsque vous créez une fonction Lambda à partir d'un modèle AWS SAM, AWS SAM crée automatiquement un package de déploiement .zip ou une image de conteneur avec le code

de votre fonction et toutes les dépendances que vous spécifiez. Ensuite, AWS SAM déploie votre fonction à l'aide d'une pile CloudFormation. Pour en savoir plus sur l'utilisation d'AWS SAM pour créer et déployer des fonctions Lambda, veuillez consulter la rubrique Premiers pas avec AWS SAM dans le Guide du développeur AWS Serverless Application Model.

Les étapes suivantes vous montrent comment télécharger, créer et déployer un exemple d'application Hello World en .NET à l'aide de AWS SAM. Cet exemple d'application utilise une fonction Lambda et un point de terminaison Amazon API Gateway pour implémenter une API backend de base. Lorsque vous envoyez une requête HTTP GET à votre point de terminaison API Gateway, API Gateway invoque votre fonction Lambda. La fonction renvoie un message « hello world », ainsi que l'adresse IP de l'instance de la fonction Lambda qui traite votre requête.

Lorsque vous créez et déployez votre application à l'aide de AWS SAM, en arrière-plan, la CLI AWS SAM utilise la commande `dotnet lambda package` pour empaqueter les différents paquets de code de la fonction Lambda.

Prérequis

Kit SDK .NET 8

Installez l'environnement d'exécution et le kit SDK .NET 8.

Version 1.39 ou ultérieure de la CLI AWS SAM

Pour savoir comment installer la dernière version de la CLI AWS SAM, consultez la section Installation de la CLI AWS SAM.

Déployer un exemple d'application AWS SAM

1. Initialisez l'application en utilisant le modèle Hello world .NET à l'aide de la commande suivante.

```
sam init --app-template hello-world --name sam-app \
--package-type Zip --runtime dotnet8
```

Cette commande crée les fichiers et répertoires suivants dans le répertoire de votre projet.

```
### sam-app
    ### README.md
    ### events
    #   ### event.json
```

```
    ### omnisharp.json
    ### samconfig.toml
    ### src
    #   ### HelloWorld
    #       ### Function.cs
    #       ### HelloWorld.csproj
    #       ### aws-lambda-tools-defaults.json
    ### template.yaml
    ### test
        ### HelloWorld.Test
            ### FunctionTest.cs
            ### HelloWorld.Tests.csproj
```

2. Accédez au répertoire contenant le `template.yaml file`. Ce fichier est un modèle qui définit les ressources AWS de votre application, notamment votre fonction Lambda et API Gateway.

```
cd sam-app
```

3. Pour créer la source de votre application, exécutez la commande suivante.

```
sam build
```

4. Pour déployer votre application sur AWS, exécutez la commande suivante.

```
sam deploy --guided
```

Cette commande permet d'empaqueter et de déployer votre application à l'aide de la série d'invites suivante. Pour accepter les options par défaut, appuyez sur Entrée.

> **Note**
>
> Pour HelloWorldFunction peut ne pas avoir d'autorisation définie, est-ce correct ?, assurez-vous de saisir y.

- Nom de la pile : nom de la pile à déployer sur CloudFormation. Ce nom doit être propre à votre Compte AWS et à votre Région AWS.
- Région AWS : la Région AWS dans laquelle vous souhaitez déployer votre application.
- Confirmer les modifications avant le déploiement : sélectionnez oui pour examiner manuellement tous les ensembles de modifications avant que AWS SAM ne déploie

les modifications de l'application. Si vous sélectionnez non, la CLI AWS SAM déploie automatiquement les modifications de l'application.

- Autoriser la création de rôles IAM par la CLI SAM : de nombreux modèles AWS SAM, y compris celui de Hello world dans cet exemple, créent des rôles (IAM) Gestion des identités et des accès AWS pour donner à vos fonctions Lambda l'autorisation d'accéder à d'autres Services AWS. Sélectionnez Oui pour autoriser le déploiement d'une pile CloudFormation qui crée ou modifie des rôles IAM.
- Désactiver le retour en arrière : par défaut, si AWS SAM rencontre une erreur lors de la création ou du déploiement de votre pile, il revient à la version précédente. Sélectionnez Non pour accepter cette valeur par défaut.
- HelloWorldFunction n'a peut-être pas d'autorisation définie, est-ce correct : Saisissez `y`.
- Enregistrer les arguments dans samconfig.toml : sélectionnez oui pour enregistrer vos choix de configuration. À l'avenir, vous pourrez à nouveau exécuter `sam deploy` sans paramètres pour déployer les modifications apportées à votre application.

5. Une fois le déploiement de votre application terminé, la CLI renvoie l'Amazon Resource Name (ARN) de la fonction Lambda Hello World et le rôle IAM créé pour cette fonction. Elle affiche également le point de terminaison de votre API Gateway. Pour tester votre application, ouvrez le point de terminaison dans un navigateur. Vous devriez voir une réponse similaire à la suivante.

```
{"message":"hello world","location":"34.244.135.203"}
```

6. Pour supprimer vos ressources, exécutez la commande suivante. Notez que le point de terminaison de l'API que vous avez créé est un point de terminaison public accessible via le réseau Internet. Nous vous recommandons de supprimer ce point de terminaison après les tests.

```
sam delete
```

Étapes suivantes

Pour en savoir plus sur l'utilisation de AWS SAM pour créer et déployer des fonctions Lambda à l'aide de .NET, consultez les ressources suivantes :

- Le Guide du développeur AWS Serverless Application Model (AWS SAM)
- Création d'applications .NET sans serveur à l'aide de AWS Lambda et de la CLI SAM

Déploiement de fonctions Lambda C# à l'aide d'AWS CDK

Le AWS Cloud Development Kit (AWS CDK) est un cadre de développement logiciel libre permettant de définir une infrastructure cloud sous forme de code à l'aide de langages de programmation modernes et de cadres tels que .NET. Les projets AWS CDK sont exécutés pour générer des modèles CloudFormation qui sont ensuite utilisés pour déployer votre code.

Pour créer et déployer un exemple d'application Hello world .NET à l'aide de AWS CDK, suivez les instructions décrites dans les sections suivantes. L'exemple d'application implémente une API backend de base composée d'un point de terminaison API Gateway et d'une fonction Lambda. Lorsque vous envoyez une requête HTTP GET au point de terminaison, API Gateway invoque la fonction Lambda. La fonction renvoie un message Hello world, ainsi que l'adresse IP de l'instance Lambda qui traite votre requête.

Prérequis

Kit SDK .NET 8

Installez l'environnement d'exécution et le kit SDK .NET 8.

Version 2 de AWS CDK

Pour savoir comment installer la dernière version de AWS CDK, consultez la section Démarrer avec le AWS CDK dans le Guide du développeur v2 de AWS Cloud Development Kit (AWS CDK).

Déployer un exemple d'application AWS CDK

1. Créez un répertoire de projet pour l'application modèle et accédez à ce répertoire.

```
mkdir hello-world
cd hello-world
```

2. Initialisez une nouvelle application AWS CDK en exécutant la commande suivante.

```
cdk init app --language csharp
```

La commande crée les fichiers et répertoires suivants dans votre répertoire de projet.

```
### README.md
### cdk.json
```

```
### src
    ### HelloWorld
    #   ### GlobalSuppressions.cs
    #   ### HelloWorld.csproj
    #   ### HelloWorldStack.cs
    #   ### Program.cs
    ### HelloWorld.sln
```

3. Ouvrez le répertoire `src` et créez une nouvelle fonction Lambda à l'aide de la CLI .NET. Il s'agit de la fonction que vous allez déployer à l'aide de AWS CDK. Dans cet exemple, vous créez une fonction Hello world nommée `HelloWorldLambda` à l'aide du modèle `lambda.EmptyFunction`.

```
cd src
dotnet new lambda.EmptyFunction -n HelloWorldLambda
```

Après cette étape, la structure de votre répertoire à l'intérieur de votre répertoire de projet devrait ressembler à ce qui suit.

```
### README.md
### cdk.json
### src
    ### HelloWorld
    #   ### GlobalSuppressions.cs
    #   ### HelloWorld.csproj
    #   ### HelloWorldStack.cs
    #   ### Program.cs
    ### HelloWorld.sln
    ### HelloWorldLambda
        ### src
        #   ### HelloWorldLambda
        #       ### Function.cs
        #       ### HelloWorldLambda.csproj
        #       ### Readme.md
        #       ### aws-lambda-tools-defaults.json
        ### test
            ### HelloWorldLambda.Tests
                ### FunctionTest.cs
                ### HelloWorldLambda.Tests.csproj
```

4. Ouvrez le fichier `HelloWorldStack.cs` à partir du répertoire `src/HelloWorld`. Remplacez le contenu du fichier par le code suivant.

```
using Amazon.CDK;
using Amazon.CDK.AWS.Lambda;
using Amazon.CDK.AWS.Logs;
using Constructs;

namespace CdkTest
{
    public class HelloWorldStack : Stack
    {
        internal HelloWorldStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            var buildOption = new BundlingOptions()
            {
                Image = Runtime.DOTNET_8.BundlingImage,
                User = "root",
                OutputType = BundlingOutput.ARCHIVED,
                Command = new string[]{
            "/bin/sh",
                "-c",
                " dotnet tool install -g Amazon.Lambda.Tools"+
                " && dotnet build"+
                " && dotnet lambda package --output-package /asset-output/
function.zip"
                }
            };

             var helloWorldLambdaFunction = new Function(this,
 "HelloWorldFunction", new FunctionProps
            {
                Runtime = Runtime.DOTNET_8,
                MemorySize = 1024,
                LogRetention = RetentionDays.ONE_DAY,
                Handler =
 "HelloWorldLambda::HelloWorldLambda.Function::FunctionHandler",
                Code = Code.FromAsset("./src/HelloWorldLambda/src/
HelloWorldLambda", new Amazon.CDK.AWS.S3.Assets.AssetOptions
                {
                    Bundling = buildOption
                }),
            });
        }
    }
```

```
}
```

Il s'agit du code permettant de compiler et de regrouper le code de l'application, ainsi que de la définition de la fonction Lambda elle-même. L'objet `BundlingOptions` permet de créer un fichier zip, ainsi qu'un ensemble de commandes utilisées pour générer le contenu du fichier zip. Dans ce cas, la commande `dotnet lambda package` est utilisée pour compiler et générer le fichier zip.

5. Pour déployer votre application, exécutez la commande suivante.

```
cdk deploy
```

6. Invoquez votre fonction Lambda déployée à l'aide de la CLI .NET Lambda.

```
dotnet lambda invoke-function HelloWorldFunction -p "hello world"
```

7. Une fois les tests terminés, vous pouvez supprimer les ressources que vous avez créées, à moins que vous souhaitez les conserver. Exécutez la commande suivante pour supprimer vos ressources.

```
cdk destroy
```

Étapes suivantes

Pour en savoir plus sur l'utilisation de AWS CDK pour créer et déployer des fonctions Lambda à l'aide de .NET, consultez les ressources suivantes :

- Utilisation de CDK AWS en C#
- Créez, empaquetez et publiez des fonctions Lambda .NET C# à l'aide de CDK AWS

Déployez des applications ASP.NET

Outre l'hébergement de fonctions événementielles, vous pouvez également utiliser .NET avec Lambda pour héberger des applications ASP.NET légères. Vous pouvez créer et déployer des applications ASP.NET à l'aide du package NuGet `Amazon.Lambda.AspNetCoreServer`. Dans cette section, vous apprendrez à déployer une API web ASP.NET sur Lambda à l'aide de l'outillage de la CLI .NET Lambda.

Rubriques

- Prérequis
- Déploiement d'une API Web ASP.NET sur Lambda
- Déploiement des API minimales ASP.NET sur Lambda

Prérequis

Kit SDK .NET 8

Installez l'environnement d'exécution ASP.NET Core et le kit SDK .NET 8.

Amazon.Lambda.Tools

Pour créer vos fonctions Lambda, utilisez l'Amazon.Lambda.Tools extension outils globaux .NET. Pour installer Amazon.Lambda.Tools, exécutez la commande suivante :

```
dotnet tool install -g Amazon.Lambda.Tools
```

Pour en savoir plus sur l'extension CLI .NET Amazon.Lambda.Tools, consultez le référentiel Extensions pour la CLI .NET AWS dans GitHub.

Amazon.Lambda.Templates

Pour générer votre code de fonction Lambda, utilisez le package NuGet Amazon.Lambda.Templates. Pour installer ce package de modèle, exécutez la commande suivante :

```
dotnet new --install Amazon.Lambda.Templates
```

Déploiement d'une API Web ASP.NET sur Lambda

Pour déployer une API web à l'aide d'ASP.NET, vous pouvez utiliser les modèles Lambda .NET pour créer un nouveau projet d'API web. Utilisez la commande suivante pour initialiser un nouveau projet d'API web ASP.NET. Dans l'exemple de commande, nous nommons le projet `AspNetOnLambda`.

```
dotnet new serverless.AspNetCoreWebAPI -n AspNetOnLambda
```

Cette commande crée les fichiers et répertoires suivants dans le répertoire de votre projet.

```
.
### AspNetOnLambda
    ### src
    #   ### AspNetOnLambda
    #       ### AspNetOnLambda.csproj
    #       ### Controllers
    #       #   ### ValuesController.cs
    #       ### LambdaEntryPoint.cs
    #       ### LocalEntryPoint.cs
    #       ### Readme.md
    #       ### Startup.cs
    #       ### appsettings.Development.json
    #       ### appsettings.json
    #       ### aws-lambda-tools-defaults.json
    #       ### serverless.template
    ### test
        ### AspNetOnLambda.Tests
            ### AspNetOnLambda.Tests.csproj
            ### SampleRequests
            #   ### ValuesController-Get.json
            ### ValuesControllerTests.cs
            ### appsettings.json
```

Lorsque Lambda invoque votre fonction, le point d'entrée qu'elle utilise est le fichier `LambdaEntryPoint.cs`. Le fichier créé par le modèle Lambda .NET contient le code suivant.

```
namespace AspNetOnLambda;

public class LambdaEntryPoint : Amazon.Lambda.AspNetCoreServer.APIGatewayProxyFunction
{
    protected override void Init(IWebHostBuilder builder)
    {
        builder
            .UseStartup#Startup#();
    }

    protected override void Init(IHostBuilder builder)
    {
    }
}
```

Le point d'entrée utilisé par Lambda doit hériter de l'une des trois classes de base du package `Amazon.Lambda.AspNetCoreServer`. Ces trois classes de base sont les suivantes :

- `APIGatewayProxyFunction`
- `APIGatewayHttpApiV2ProxyFunction`
- `ApplicationLoadBalancerFunction`

La classe par défaut utilisée lorsque vous créez votre fichier `LambdaEntryPoint.cs` à l'aide du modèle Lambda .NET fourni est `APIGatewayProxyFunction`. La classe de base que vous utilisez dans votre fonction dépend de la couche d'API qui se trouve devant votre fonction Lambda.

Chacune des trois classes de base contient une méthode publique appelée `FunctionHandlerAsync`. Le nom de cette méthode fera partie de la chaîne du gestionnaire que Lambda utilise pour invoquer votre fonction. La méthode `FunctionHandlerAsync` transforme la charge utile de l'événement entrant dans le format ASP.NET correct et la réponse ASP.NET en une charge utile de réponse Lambda. Pour l'exemple de projet `AspNetOnLambda` affiché, la chaîne du gestionnaire serait la suivante.

```
AspNetOnLambda::AspNetOnLambda.LambdaEntryPoint::FunctionHandlerAsync
```

Pour déployer l'API sur Lambda, exécutez les commandes suivantes pour accéder au répertoire contenant votre fichier de code source et déployer votre fonction à l'aide de CloudFormation.

```
cd AspNetOnLambda/src/AspNetOnLambda
dotnet lambda deploy-serverless
```

> **Tip**
>
> Lorsque vous déployez une API à l'aide de la commande **`dotnet lambda deploy-serverless`**, CloudFormation attribue à votre fonction Lambda un nom basé sur le nom de pile que vous avez spécifié lors du déploiement. Pour donner un nom personnalisé à votre fonction Lambda, modifiez le fichier `serverless.template` pour ajouter une propriété `FunctionName` à la ressource `AWS::Serverless::Function`. Pour en savoir plus, consultez Type de nom dans le Guide de l'utilisateur CloudFormation.

Déploiement des API minimales ASP.NET sur Lambda

Pour déployer une API minimale ASP.NET sur Lambda, vous pouvez utiliser les modèles Lambda .NET pour créer un nouveau projet d'API minimale. Utilisez la commande suivante pour initialiser un nouveau projet d'API minimale. Dans cet exemple, nous nommons le projet `MinimalApiOnLambda`.

```
dotnet new serverless.AspNetCoreMinimalAPI -n MinimalApiOnLambda
```

La commande crée les fichiers et répertoires suivants dans le répertoire de votre projet.

```
### MinimalApiOnLambda
    ### src
        ### MinimalApiOnLambda
            ### Controllers
            #   ### CalculatorController.cs
            ### MinimalApiOnLambda.csproj
            ### Program.cs
            ### Readme.md
            ### appsettings.Development.json
            ### appsettings.json
            ### aws-lambda-tools-defaults.json
            ### serverless.template
```

Le fichier `Program.cs` contient le code suivant.

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllers();

// Add AWS Lambda support. When application is run in Lambda Kestrel is swapped out as
 the web server with Amazon.Lambda.AspNetCoreServer. This
// package will act as the webserver translating request and responses between the
 Lambda event source and ASP.NET Core.
builder.Services.AddAWSLambdaHosting(LambdaEventSource.RestApi);

var app = builder.Build();


app.UseHttpsRedirection();
app.UseAuthorization();
```

```
app.MapControllers();

app.MapGet("/", () => "Welcome to running ASP.NET Core Minimal API on AWS Lambda");

app.Run();
```

Pour configurer votre API minimale afin qu'elle s'exécute sur Lambda, vous devrez peut-être modifier ce code pour que les demandes et les réponses entre Lambda et ASP.NET Core soient correctement traduites. La fonction est configurée par défaut pour une source d'événements d'API REST. Pour une API HTTP ou un équilibreur de charge d'application, remplacez `(LambdaEventSource.RestApi)` par l'une des options suivantes :

- `(LambdaEventSource.HttpAPi)`
- `(LambdaEventSource.ApplicationLoadBalancer)`

Pour déployer votre API minimale sur Lambda, exécutez les commandes suivantes pour accéder au répertoire contenant votre fichier de code source et déployer votre fonction à l'aide de CloudFormation.

```
cd MinimalApiOnLambda/src/MinimalApiOnLambda
dotnet lambda deploy-serverless
```

Utilisation de couches pour les fonctions Lambda .NET

Nous déconseillons d'utiliser des couches pour gérer les dépendances des fonctions Lambda écrites en .NET. .NET étant un langage compilé, vos fonctions doivent toujours charger manuellement les assemblages partagés dans la mémoire pendant la phase d'initialisation, ce qui peut augmenter les temps de démarrage à froid. L'utilisation de couches complique non seulement le processus de déploiement, mais vous empêche également de tirer parti des optimisations du compilateur intégré.

Pour utiliser des dépendances externes avec vos gestionnaires .NET, incluez-les directement dans votre package de déploiement au moment de la compilation. Ce faisant, vous simplifiez le processus de déploiement et profitez des optimisations du compilateur .NET intégré. Pour un exemple d'importation et d'utilisation des dépendances comme les packages NuGet dans votre fonction, consultez the section called "Handler (Gestionnaire)".

Déployer des fonctions Lambda .NET avec des images conteneurs

Il existe trois méthodes pour créer une image de conteneur pour une fonction Lambda .NET :

- Utilisation d'une image AWS de base pour .NET

  Les images de base AWS sont préchargées avec une exécution du langage, un client d'interface d'exécution pour gérer l'interaction entre Lambda et votre code de fonction, et un émulateur d'interface d'exécution pour les tests locaux.

- Utilisation d'une image de base AWS uniquement pour le système d'exploitation

  AWS Les images de base réservées au système d'exploitation contiennent une distribution Amazon Linux et l'émulateur d'interface d'exécution. Ces images sont couramment utilisées pour créer des images de conteneur pour les langages compilés, tels que Go et Rust, et pour une langue ou une version linguistique pour laquelle Lambda ne fournit pas d'image de base, comme Node.js 19. Vous pouvez également utiliser des images de base uniquement pour le système d'exploitation pour implémenter un environnement d'exécution personnalisé. Pour rendre l'image compatible avec Lambda, vous devez inclure le client d'interface d'exécution pour .NET dans l'image.

- Utilisation d'une image non AWS basique

  Vous pouvez utiliser une autre image de base à partir d'un autre registre de conteneur, comme Alpine Linux ou Debian. Vous pouvez également utiliser une image personnalisée créée par votre organisation. Pour rendre l'image compatible avec Lambda, vous devez inclure le client d'interface d'exécution pour .NET dans l'image.

> **Tip**
>
> Pour réduire le temps nécessaire à l'activation des fonctions du conteneur Lambda, consultez Utiliser des générations en plusieurs étapes (français non garanti) dans la documentation Docker. Pour créer des images de conteneur efficaces, suivez la section Bonnes pratiques pour l'écriture de Dockerfiles (français non garanti).

Cette page explique comment créer, tester et déployer des images de conteneur pour Lambda.

Rubriques

- AWS images de base pour .NET
- Utilisation d'une image AWS de base pour .NET
- Utilisation d'une autre image de base avec le client d'interface d'exécution

AWS images de base pour .NET

AWS fournit les images de base suivantes pour .NET :

Étiquettes	Environnement d'exécution	Système d'exploitation	Dockerfile	Obsolescence
10	.NET 10	Amazon Linux	Dockerfile pour .NET 10 sur GitHub	14 novembre 2028
9	.NET 9	Amazon Linux	Dockerfile pour .NET 9 sur GitHub	10 novembre 2026
8	.NET 8	Amazon Linux	Dockerfile pour .NET 8 sur GitHub	10 novembre 2026

Référentiel Amazon ECR : gallery.ecr. aws/lambda/dotnet

Utilisation d'une image AWS de base pour .NET

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- SDK .NET : les étapes suivantes utilisent l'image de base .NET 8. Assurez-vous que votre version de .NET correspond à la version de l'image de base que vous spécifiez dans votre Dockerfile.
- Docker (version minimale 25.0.0)
- Le plug-in buildx Docker.

Création et déploiement d'une image à l'aide d'une image de base

Dans les étapes suivantes, vous utilisez Amazon.Lambda.Templates et Amazon.Lambda.Tools pour créer un projet .NET. Ensuite, vous créez une image Docker, vous chargez l'image sur Amazon ECR et vous la déployez vers une fonction Lambda.

1. Installez le package Amazon.Lambda.Templates. NuGet

   ```
   dotnet new install Amazon.Lambda.Templates
   ```

2. Créez un projet .NET en utilisant le modèle `lambda.image.EmptyFunction`.

   ```
   dotnet new lambda.image.EmptyFunction --name MyFunction --region us-east-1
   ```

   Les fichiers de projet sont stockés dans le répertoire *`MyFunction`*`/src/`*`MyFunction`* :

   - aws-lambda-tools-defaults.json : Spécifie les options de ligne de commande pour déployer votre fonction Lambda.
   - Function.cs : code de fonction de votre gestionnaire Lambda. Il s'agit d'un modèle C# qui inclut la bibliothèque `Amazon.Lambda.Core` par défaut et un attribut `LambdaSerializer` par défaut. Pour plus d'informations sur les conditions de sérialisation et les options, consultez Sérialisation dans les fonctions Lambda C#. Vous pouvez utiliser le code fourni pour les tests ou le remplacer par le vôtre.
   - MyFunction.csproj : fichier de projet .NET répertoriant les fichiers et les assemblages composant votre application.
   - Dockerfile : vous pouvez utiliser le Dockerfile fourni pour les tests ou le remplacer par le vôtre. Si vous utilisez le vôtre, assurez-vous de :
     - Définir la propriété `FROM` sur l'URI de l'image de base. L'image de base et `TargetFramework` dans le fichier `MyFunction.csproj` doivent tous deux utiliser la même version de .NET. Par exemple, pour utiliser .NET 9 :
       - Fichier Docker : `FROM` *`public.ecr.aws/lambda/dotnet:9`*
       - MyFunction.csproj : `<TargetFramework>`*`net9.0`*`</TargetFramework>`
     - Définir l'argument `CMD` pour le gestionnaire de la fonction Lambda. Il doit correspondre à `image-command` dans `aws-lambda-tools-defaults.json`.

3. Installer l'outil global .NET d'Amazon.Lambda.Tools.

```
dotnet tool install -g Amazon.Lambda.Tools
```

Si Amazon.Lambda.Tools est déjà installé, assurez-vous que vous disposez de la dernière version.

```
dotnet tool update -g Amazon.Lambda.Tools
```

4. Changez le répertoire en *MyFunction*/src/*MyFunction*, si vous n'y êtes pas déjà.

```
cd src/MyFunction
```

5. Utilisez Amazon.Lambda.Tools pour créer l'image Docker, l'envoyer (push) vers un nouveau référentiel Amazon ECR, et déployer la fonction Lambda.

   Pour `--function-role`, indiquez le nom du rôle, et non l'Amazon Resource Name (ARN), du rôle d'exécution de la fonction. Par exemple, `lambda-role`.

```
dotnet lambda deploy-function MyFunction --function-role lambda-role
```

   Pour plus d'informations sur l'outil global .NET Amazon.Lambda.Tools, consultez le référentiel Extensions AWS for .NET CLI sur. GitHub

6. Invoquer la fonction.

```
dotnet lambda invoke-function MyFunction --payload "Testing the function"
```

   Si tout réussit, vous verrez une réponse similaire à ce qui suit :

```
Payload:
{"Lower":"testing the function","Upper":"TESTING THE FUNCTION"}

Log Tail:
INIT_REPORT Init Duration: 9999.81 ms   Phase: init     Status: timeout
START RequestId: 12378346-f302-419b-b1f2-deaa1e8423ed Version: $LATEST
END RequestId: 12378346-f302-419b-b1f2-deaa1e8423ed
REPORT RequestId: 12378346-f302-419b-b1f2-deaa1e8423ed  Duration: 3173.06 ms
 Billed Duration: 3174 ms        Memory Size: 512 MB     Max Memory Used: 24 MB
```

7. Supprimez la fonction Lambda.

```
dotnet lambda delete-function MyFunction
```

Utilisation d'une autre image de base avec le client d'interface d'exécution

Si vous utilisez une image de base uniquement pour le système d'exploitation ou une autre image de base, vous devez inclure le client d'interface d'exécution dans votre image. Le client d'interface d'exécution étend leAPI de runtime, qui gère l'interaction entre Lambda et votre code de fonction.

L'exemple suivant montre comment créer une image de conteneur pour .NET à l'aide d'une image non AWS basique et comment ajouter le fichier Amazon.Lambda. RuntimeSupport package, qui est le client d'interface d'exécution Lambda pour .NET. L'exemple de Dockerfile utilise l'image de base Microsoft .NET 8.

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- SDK .NET — Les étapes suivantes utilisent une image de base .NET 9. Assurez-vous que votre version de .NET correspond à la version de l'image de base que vous spécifiez dans votre Dockerfile.
- Docker (version minimale 25.0.0)
- Le plug-in buildx Docker.

Création et déploiement d'une image à l'aide d'une image de base alternative

1. Installez le package Amazon.Lambda.Templates. NuGet

   ```
   dotnet new install Amazon.Lambda.Templates
   ```

2. Créez un projet .NET en utilisant le modèle `lambda.CustomRuntimeFunction`. Ce modèle inclut l'Amazon.Lambda. RuntimeSupportcolis.

   ```
   dotnet new lambda.CustomRuntimeFunction --name MyFunction --region us-east-1
   ```

3. Accédez au répertoire *MyFunction*`/src/`*MyFunction*. C'est ici que les fichiers du projet sont stockés. Examinez les fichiers suivants :

- aws-lambda-tools-defaults.json — Ce fichier vous permet de spécifier les options de ligne de commande lors du déploiement de votre fonction Lambda.
- Function.cs : le code contient une classe avec une méthode `Main` qui initialise la bibliothèque `Amazon.Lambda.RuntimeSupport` en tant qu'amorce. La méthode `Main` est le point d'entrée du processus de la fonction. La méthode `Main` enveloppe le gestionnaire de fonctions dans un encapsuleur avec lequel l'amorce peut travailler. Pour plus d'informations, consultez la section Utilisation d'Amazon.Lambda. RuntimeSupport en tant que bibliothèque de classes dans le GitHub référentiel.
- MyFunction.csproj : fichier de projet .NET répertoriant les fichiers et les assemblages composant votre application.
- Readme.md : ce fichier contient plus d'informations sur l'exemple de fonction Lambda.

4. Ouvrez le fichier `aws-lambda-tools-defaults.json` et ajoutez les lignes suivantes :

```
  "package-type": "image",
  "docker-host-build-output-dir": "./bin/Release/lambda-publish"
```

- package-typee : définit le package de déploiement en tant qu'image de conteneur.
- docker-host-build-output-dir : définit le répertoire de sortie pour le processus de construction.

Example aws-lambda-tools-defaults.json

```
{
  "Information": [
    "This file provides default values for the deployment wizard inside Visual
 Studio and the AWS Lambda commands added to the .NET Core CLI.",
    "To learn more about the Lambda commands with the .NET Core CLI execute the
 following command at the command line in the project root directory.",
    "dotnet lambda help",
    "All the command line options for the Lambda command can be specified in this
 file."
  ],
  "profile": "",
  "region": "us-east-1",
  "configuration": "Release",
  "function-runtime": "provided.al2023",
  "function-memory-size": 256,
  "function-timeout": 30,
```

```
  "function-handler": "bootstrap",
  "msbuild-parameters": "--self-contained true",
  "package-type": "image",
  "docker-host-build-output-dir": "./bin/Release/lambda-publish"
}
```

5. Créez un Dockerfile dans le référentiel *MyFunction*/src/*MyFunction*. L'exemple de Dockerfile suivant utilise une image de base Microsoft .NET au lieu d'une image de base AWS.

   - Définissez la propriété FROM pour l'identifiant de l'image de base. L'image de base et TargetFramework dans le fichier MyFunction.csproj doivent tous deux utiliser la même version de .NET.
   - Utilisez la commande COPY pour copier la fonction dans le répertoire /var/task.
   - Définissez le ENTRYPOINT sur le module que vous souhaitez que le conteneur Docker exécute lorsqu'il démarre. Dans ce cas, le module est l'amorce, qui initialise la bibliothèque Amazon.Lambda.RuntimeSupport.

   Notez que l'exemple de Dockerfile n'inclut pas d'instruction USER. Lorsque vous déployez une image de conteneur sur Lambda, Lambda définit automatiquement un utilisateur Linux par défaut disposant d'autorisations de moindre privilège. Ceci est différent du comportement standard de Docker qui est défini par défaut par l'utilisateur root lorsqu'aucune instruction USER n'est fournie.

   Example Dockerfile

```
# You can also pull these images from DockerHub amazon/aws-lambda-dotnet:8
FROM mcr.microsoft.com/dotnet/runtime:9.0

# Set the image's internal work directory
WORKDIR /var/task

# Copy function code to Lambda-defined environment variable
COPY "bin/Release/net9.0/linux-x64"  .

# Set the entrypoint to the bootstrap
ENTRYPOINT ["/usr/bin/dotnet", "exec", "/var/task/bootstrap.dll"]
```

6. Installez l'extension de l'outil global .NET d'Amazon.Lambda.Tools.

```
dotnet tool install -g Amazon.Lambda.Tools
```

Si Amazon.Lambda.Tools est déjà installé, assurez-vous que vous disposez de la dernière version.

```
dotnet tool update -g Amazon.Lambda.Tools
```

7. Utilisez Amazon.Lambda.Tools pour créer l'image Docker, l'envoyer (push) vers un nouveau référentiel Amazon ECR, et déployer la fonction Lambda.

   Pour `--function-role`, indiquez le nom du rôle, et non l'Amazon Resource Name (ARN), du rôle d'exécution de la fonction. Par exemple, `lambda-role`.

   ```
   dotnet lambda deploy-function MyFunction --function-role lambda-role
   ```

   Pour plus d'informations sur l'extension .NET CLI Amazon.Lambda.Tools, consultez le référentiel AWS Extensions for .NET CLI sur. GitHub
8. Invoquer la fonction.

   ```
   dotnet lambda invoke-function MyFunction --payload "Testing the function"
   ```

   Si tout réussit, vous devez voir ce qui suit :

   ```
   Payload:
   "TESTING THE FUNCTION"

   Log Tail:
   START RequestId: id Version: $LATEST
   END RequestId: id
   REPORT RequestId: id  Duration: 0.99 ms       Billed Duration: 1 ms         Memory
    Size: 256 MB     Max Memory Used: 12 MB
   ```
9. Supprimez la fonction Lambda.

   ```
   dotnet lambda delete-function MyFunction
   ```

Compilation du code de fonction Lambda .NET dans un format d'exécution natif

.NET 8 prend en charge la compilation anticipée (AOT) native. Avec la compilation anticipée native, vous pouvez compiler le code de votre fonction Lambda dans un format d'environnement d'exécution natif, ce qui élimine la nécessité de compiler le code .NET au moment de l'exécution. La compilation anticipée native peut réduire le temps de démarrage à froid des fonctions Lambda que vous écrivez en .NET. Pour plus d'informations, consultez Introducing the .NET 8 runtime for AWS Lambda sur le blog AWS Compute.

Sections

- Le fichier d'exécution Lambda
- Prérequis
- Prise en main
- Sérialisation
- Réduction
- Résolution des problèmes

Le fichier d'exécution Lambda

Pour déployer une fonction Lambda créée avec une compilation AOT native, utilisez l'environnement d'exécution Lambda géré de .NET 8. Cet environnement d'exécution prend en charge l'utilisation des architectures x86_64 et arm64.

Lorsque vous déployez une fonction Lambda .NET sans utiliser AOT, votre application est d'abord compilée en code de langage intermédiaire (IL). Au moment de l'environnement d'exécution, le compilateur JIT (just-in-time) de l'environnement d'exécution Lambda prend le code IL et le compile en code machine selon les besoins. Avec une fonction Lambda compilée à l'avance avec l'AOT native, vous compilez votre code en code machine lorsque vous déployez votre fonction, de sorte que vous ne dépendez pas de l'environnement d'exécution .NET ou du kit SDK dans l'environnement d'exécution Lambda pour compiler votre code avant qu'il ne s'exécute.

L'une des limites de l'AOT est que le code de votre application doit être compilé dans un environnement doté du même système d'exploitation Amazon Linux 2023 (AL2023) que celui utilisé par l'environnement d'exécution .NET 8. La CLI Lambda .NET fournit une fonctionnalité permettant de compiler votre application dans un conteneur Docker à l'aide d'une image AL2023.

Pour éviter d'éventuels problèmes de compatibilité entre architectures, nous vous recommandons vivement de compiler votre code dans un environnement doté de la même architecture de processeur que celle que vous avez configurée pour votre fonction. Pour en savoir plus sur les limites de la compilation entre architectures, consultez Compilation croisée dans la documentation Microsoft .NET.

Prérequis

Docker

Pour utiliser l'AOT native, votre code de fonction doit être compilé dans un environnement doté du même système d'exploitation AL2023 que l'environnement d'exécution .NET 8. Les commandes de la CLI .NET des sections suivantes utilisent Docker pour développer et construire des fonctions Lambda dans un environnement AL2023.

Kit SDK .NET 8

La compilation AOT native est une fonctionnalité de .NET 8. Vous devez installer le kit SDK .NET 8 sur votre machine de compilation, et pas seulement l'environnement d'exécution.

Amazon.Lambda.Tools

Pour créer vos fonctions Lambda, utilisez l'Amazon.Lambda.Tools extension outils globaux .NET. Pour installer Amazon.Lambda.Tools, exécutez la commande suivante :

```
dotnet tool install -g Amazon.Lambda.Tools
```

Pour en savoir plus sur l'extension CLI .NET Amazon.Lambda.Tools, consultez le référentiel Extensions pour la CLI .NET AWS dans GitHub.

Amazon.Lambda.Templates

Pour générer votre code de fonction Lambda, utilisez le package NuGet Amazon.Lambda.Templates. Pour installer ce package de modèle, exécutez la commande suivante :

```
dotnet new install Amazon.Lambda.Templates
```

Prise en main

La CLI globale .NET et le AWS Serverless Application Model (AWS SAM) fournissent tous deux des modèles de démarrage pour créer des applications à l'aide de la compilation anticipée native. Pour

créer votre première fonction Lambda à l'aide de la compilation anticipée native, suivez les étapes décrites dans les instructions suivantes.

Pour initialiser et déployer une fonction Lambda compilée à l'aide de la compilation anticipée native

1. Initialisez un nouveau projet en utilisant le modèle de la compilation anticipée native, puis naviguez dans le répertoire contenant les fichiers créés `.cs` et `.csproj`. Dans cet exemple, nous allons nommer notre fonction `NativeAotSample`.

   ```
   dotnet new lambda.NativeAOT -n NativeAotSample
   cd ./NativeAotSample/src/NativeAotSample
   ```

   Le fichier créé `Function.cs` par le modèle de la compilation anticipée native contient le code de fonction suivant.

   ```
   using Amazon.Lambda.Core;
   using Amazon.Lambda.RuntimeSupport;
   using Amazon.Lambda.Serialization.SystemTextJson;
   using System.Text.Json.Serialization;

   namespace NativeAotSample;

   public class Function
   {
       /// <summary>
       /// The main entry point for the Lambda function. The main function is called
    once during the Lambda init phase. It
       /// initializes the .NET Lambda runtime client passing in the function handler
    to invoke for each Lambda event and
       /// the JSON serializer to use for converting Lambda JSON format to the .NET
    types.
       /// </summary>
       private static async Task Main()
       {
           Func<string, ILambdaContext, string> handler = FunctionHandler;
           await LambdaBootstrapBuilder.Create(handler, new
    SourceGeneratorLambdaJsonSerializer<LambdaFunctionJsonSerializerContext>())
               .Build()
               .RunAsync();
       }

       /// <summary>
   ```

```
    /// A simple function that takes a string and does a ToUpper.
    ///
    /// To use this handler to respond to an AWS event, reference the appropriate
 package from
    /// https://github.com/aws/aws-lambda-dotnet#events
    /// and change the string input parameter to the desired event type. When the
 event type
    /// is changed, the handler type registered in the main method needs to be
 updated and the LambdaFunctionJsonSerializerContext
    /// defined below will need the JsonSerializable updated. If the return type
 and event type are different then the
    /// LambdaFunctionJsonSerializerContext must have two JsonSerializable
 attributes, one for each type.
    ///
    // When using Native AOT extra testing with the deployed Lambda functions is
 required to ensure
    // the libraries used in the Lambda function work correctly with Native AOT. If
 a runtime
    // error occurs about missing types or methods the most likely solution will be
 to remove references to trim-unsafe
    // code or configure trimming options. This sample defaults to partial TrimMode
 because currently the AWS
    // SDK for .NET does not support trimming. This will result in a larger
 executable size, and still does not
    // guarantee runtime trimming errors won't be hit.
    /// </summary>
    /// <param name="input"></param>
    /// <param name="context"></param>
    /// <returns></returns>
    public static string FunctionHandler(string input, ILambdaContext context)
    {
        return input.ToUpper();
    }
}

/// <summary>
/// This class is used to register the input event and return type for the
 FunctionHandler method with the System.Text.Json source generator.
/// There must be a JsonSerializable attribute for each type used as the input and
 return type or a runtime error will occur
/// from the JSON serializer unable to find the serialization information for
 unknown types.
/// </summary>
[JsonSerializable(typeof(string))]
```

```
public partial class LambdaFunctionJsonSerializerContext : JsonSerializerContext
{
    // By using this partial class derived from JsonSerializerContext, we can
 generate reflection free JSON Serializer code at compile time
    // which can deserialize our class and properties. However, we must attribute
 this class to tell it what types to generate serialization code for.
    // See https://docs.microsoft.com/en-us/dotnet/standard/serialization/system-
text-json-source-generation
```

La compilation anticipée native permet de compiler votre application en un seul fichier binaire natif. Le point d'entrée de ce fichier binaire est la méthode `static Main`. Dans `static Main`, l'exécution Lambda est amorcée et la méthode `FunctionHandler` est configurée. Dans le cadre de l'amorçage de l'exécution, un sérialiseur généré par la source est configuré à l'aide de `new SourceGeneratorLambdaJsonSerializer<LambdaFunctionJsonSerializerContext>()`

2. Pour déployer votre application sur Lambda, assurez-vous que Docker est en cours d'exécution dans votre environnement local et exécutez la commande suivante.

```
dotnet lambda deploy-function
```

En arrière plan, la CLI globale .NET télécharge une image Docker AL2023 et compile le code de votre application à l'intérieur d'un conteneur en cours d'exécution. Le fichier binaire compilé est renvoyé vers votre système de fichiers local avant d'être déployé sur Lambda.

3. Testez votre fonction en exécutant la commande suivante. Remplacez `<FUNCTION_NAME>` par le nom que vous avez choisi pour votre fonction dans l'assistant de déploiement.

```
dotnet lambda invoke-function <FUNCTION_NAME> --payload "hello world"
```

La réponse de la CLI comprend des détails sur les performances pour le démarrage à froid (durée d'initialisation) et la durée totale d'exécution de l'invocation de la fonction.

4. Pour supprimer les ressources AWS que vous avez créées en suivant les étapes précédentes, exécutez la commande suivante. Remplacez `<FUNCTION_NAME>` par le nom que vous avez choisi pour votre fonction dans l'assistant de déploiement. En supprimant les ressources AWS que vous n'utilisez plus, vous évitez que des frais inutiles soient facturés à votre Compte AWS.

```
dotnet lambda delete-function <FUNCTION_NAME>
```

Sérialisation

Pour déployer des fonctions sur Lambda à l'aide de la compilation anticipée native, le code de votre fonction doit utiliser la sérialisation générée par la source. Au lieu d'utiliser la réflexion au moment de l'exécution pour rassembler les métadonnées nécessaires à l'accès aux propriétés des objets pour la sérialisation, les générateurs de sources génèrent des fichiers sources C# qui sont compilés lorsque vous créez votre application. Pour configurer correctement le sérialiseur généré par la source, assurez-vous d'inclure tous les objets d'entrée et de sortie utilisés par votre fonction, ainsi que tous les types personnalisés. Par exemple, une fonction Lambda qui reçoit des événements d'API Gateway pour une API REST et renvoie un type de `Product` personnalisé inclurait un sérialiseur défini comme suit.

```
[JsonSerializable(typeof(APIGatewayProxyRequest))]
[JsonSerializable(typeof(APIGatewayProxyResponse))]
[JsonSerializable(typeof(Product))]
public partial class CustomSerializer : JsonSerializerContext
{
}
```

Réduction

La compilation anticipée native permet de réduire le code de votre application dans le cadre de la compilation afin de s'assurer que le fichier binaire est aussi petit que possible. .NET 8 pour Lambda offre une meilleure prise en charge du découpage par rapport aux versions précédentes de .NET. La prise en charge a été ajoutée aux bibliothèques d'environnement d'exécution Lambda, au kit SDK AWS .NET, aux annotations .NET Lambda et à .NET 8 lui-même.

Ces améliorations offrent la possibilité d'éliminer les avertissements de découpage au moment de la création, mais .NET ne sera jamais totalement sûr en matière de découpage. Cela signifie que certaines parties des bibliothèques sur lesquelles votre fonction repose peuvent être supprimées lors de l'étape de compilation. Vous pouvez gérer cela en le définissant `TrimmerRootAssemblies` dans le cadre de votre fichier `.csproj`, comme indiqué dans l'exemple suivant.

```
<ItemGroup>
    <TrimmerRootAssembly Include="AWSSDK.Core" />
    <TrimmerRootAssembly Include="AWSXRayRecorder.Core" />
    <TrimmerRootAssembly Include="AWSXRayRecorder.Handlers.AwsSdk" />
    <TrimmerRootAssembly Include="Amazon.Lambda.APIGatewayEvents" />
    <TrimmerRootAssembly Include="bootstrap" />
```

```
    <TrimmerRootAssembly Include="Shared" />
</ItemGroup>
```

Notez que lorsque vous recevez un avertissement de découpage, l'ajout de la classe qui génère l'avertissement `TrimmerRootAssembly` risque de ne pas résoudre le problème. Un avertissement de découpage indique que la classe essaie d'accéder à une autre classe qui ne peut être déterminée avant l'exécution. Pour éviter les erreurs d'exécution, ajoutez cette deuxième classe à `TrimmerRootAssembly`.

Pour en savoir plus sur la gestion des avertissements de découpage, consultez Introduction aux avertissements de découpage dans la documentation Microsoft .NET.

Résolution des problèmes

Error: Cross-OS native compilation is not supported. (Erreur : la compilation native entre systèmes d'exploitation n'est pas prise en charge).

> Votre version de l'outil global .NET Core Amazon.Lambda.Tools n'est pas à jour. Mettez à jour vers la dernière version et réessayez.

Docker : l'image du système d'exploitation « linux » ne peut pas être utilisée sur cette plateforme.

> Docker sur votre système est configuré pour utiliser des conteneurs Windows. Passez aux conteneurs Linux pour exécuter l'environnement de création anticipée native.

Pour plus d'informations sur les erreurs courantes, consultez le référentiel AWS NativeAOT for .NET (français non garanti) sur GitHub.

Utilisation de l’objet de contexte Lambda pour récupérer les informations de la fonction C#

Lorsque Lambda exécute votre fonction, il transmet un objet de contexte au gestionnaire. Cet objet fournit les propriétés avec des informations sur l'appel, la fonction et l'environnement d'exécution.

Propriétés du contexte

- `FunctionName` – Nom de la fonction Lambda.
- `FunctionVersion` – Version de la fonction.
- `InvokedFunctionArn` – Amazon Resource Name (ARN) utilisé pour appeler la fonction. Indique si l’appelant a spécifié un numéro de version ou un alias.
- `MemoryLimitInMB` – Quantité de mémoire allouée à la fonction.
- `AwsRequestId` – Identifiant de la demande d’invocation.
- `LogGroupName` – Groupe de journaux pour la fonction.
- `LogStreamName` – Flux de journal de l'instance de fonction.
- `RemainingTime` (`TimeSpan`) – Nombre de millisecondes restant avant l'expiration de l'exécution.
- `Identity` – (applications mobiles) Informations sur l'identité Amazon Cognito qui a autorisé la demande.
- `ClientContext` – (applications mobiles) Contexte client fourni à Lambda par l'application client.
- `Logger` L'objet enregistreur d'événements pour la fonction.

Vous pouvez utiliser les informations contenues dans l'objet `ILambdaContext` pour générer des informations sur l'invocation de votre fonction à des fins de contrôle. Le code suivant montre comment ajouter des informations contextuelles à un cadre de journalisation structuré. Dans cet exemple, la fonction ajoute `AwsRequestId` aux sorties du journal. La fonction utilise également la propriété `RemainingTime` pour annuler une tâche en vol si le délai d'attente de la fonction Lambda est sur le point d'être atteint.

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
```

```
{
    private readonly IDatabaseRepository _repo;

    public Function()
    {
        this._repo = new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request, ILambdaContext context)
    {
        Logger.AppendKey("AwsRequestId", context.AwsRequestId);

        var id = request.PathParameters["id"];

        using var cts = new CancellationTokenSource();

        try
        {
            cts.CancelAfter(context.RemainingTime.Add(TimeSpan.FromSeconds(-1)));

            var databaseRecord = await this._repo.GetById(id, cts.Token);

            return new APIGatewayProxyResponse
            {
                StatusCode = (int)HttpStatusCode.OK,
                Body = JsonSerializer.Serialize(databaseRecord)
            };
        }
        catch (Exception ex)
        {
            return new APIGatewayProxyResponse
            {
                StatusCode = (int)HttpStatusCode.InternalServerError,
                Body = JsonSerializer.Serialize(new { error = ex.Message })
            };
        }
        finally
        {
            cts.Cancel();
        }
    }
}
```

Journalisation et surveillance des fonctions Lambda C#

AWS Lambda surveille automatiquement les fonctions Lambda et envoie des entrées de journal à Amazon. CloudWatch Votre fonction Lambda est fournie avec un groupe de CloudWatch journaux Logs et un flux de journaux pour chaque instance de votre fonction. L'environnement d'exécution Lambda envoie des détails sur chaque invocation et d'autres sorties provenant du code de votre fonction au flux de journaux. Pour plus d'informations sur CloudWatch les journaux, consultezEnvoi automatique des journaux de la fonction Lambda à CloudWatch Logs.

Sections

- Création d'une fonction qui renvoie des journaux
- Utilisation des contrôles de journalisation avancés de Lambda avec .NET
- Outils de journalisation et bibliothèques supplémentaires
- Utilisation de Powertools pour AWS Lambda (.NET) et AWS SAM pour la journalisation structurée
- Affichage des journaux dans la console Lambda
- Afficher les journaux dans la CloudWatch console
- Afficher les journaux à l'aide de AWS Command Line Interface (AWS CLI)
- Suppression de journaux

Création d'une fonction qui renvoie des journaux

Pour générer des journaux à partir de votre code de fonction, vous pouvez utiliser le ILambdaLogger sur l'objet de contexte, les méthodes de la classe Console ou toute bibliothèque de journalisation qui écrit dans `stdout` ou`stderr`.

L'environnement d'exécution .NET enregistre les lignes `START`, `END` et `REPORT` pour chaque invocation. La ligne de rapport fournit les détails suivants.

Champs de données de la ligne REPORT

- RequestId— L'identifiant de demande unique pour l'invocation.
- Duration – Temps que la méthode de gestion du gestionnaire de votre fonction a consacré au traitement de l'événement.
- Billed Duration : temps facturé pour l'invocation.
- Memory Size – Quantité de mémoire allouée à la fonction.

- Max Memory Used – Quantité de mémoire utilisée par la fonction. Lorsque les appels partagent un environnement d'exécution, Lambda indique la mémoire maximale utilisée pour toutes les invocations. Ce comportement peut entraîner une valeur signalée plus élevée que prévu.
- Init Duration : pour la première requête servie, temps qu'il a pris à l'exécution charger la fonction et exécuter le code en dehors de la méthode du gestionnaire.
- XRAY TraceId — Pour les demandes suivies, l'ID de AWS X-Ray trace.
- SegmentId— Pour les demandes tracées, l'identifiant du segment X-Ray.
- Sampled – Pour les demandes suivies, résultat de l'échantillonnage.

Utilisation des contrôles de journalisation avancés de Lambda avec .NET

Pour mieux contrôler la manière dont les journaux de vos fonctions sont capturés, traités et consommés, vous pouvez configurer les options de journalisation suivantes pour les environnements d'exécution .NET pris en charge :

- Format de journal : choisissez entre le format texte brut et le format JSON structuré pour les journaux de votre fonction
- Niveau du journal : pour les journaux au format JSON, choisissez le niveau de détail des journaux auxquels Lambda envoie CloudWatch, par exemple ERROR, DEBUG ou INFO
- Groupe de journaux : choisissez le groupe de CloudWatch journaux auquel votre fonction envoie les journaux

Pour plus d'informations sur ces options de journalisation et pour savoir comment configurer votre fonction pour les utiliser, consultez the section called "Configuration de commandes de journalisation avancées pour votre fonction Lambda".

Pour utiliser le format de journal et les options de niveau de journal avec vos fonctions Lambda .NET, consultez les instructions des sections suivantes.

Utilisation du format de journal JSON structuré avec .NET

Si vous sélectionnez JSON pour le format de journal de votre fonction, Lambda enverra les journaux en sortie à l'aide de ILambdaLogger sous forme de JSON structuré. Chaque objet de journal JSON contient au moins cinq paires clé-valeur avec les clés suivantes :

- `"timestamp"` - heure à laquelle le message de journal a été généré

- `"level"` - niveau de journalisation attribué au message
- `"requestId"` - identifiant unique de la demande pour l'invocation de la fonction
- `"traceId"` : la variable d'environnement `_X_AMZN_TRACE_ID`
- `"message"` - contenu du message de journal

L'instance `ILambdaLogger` peut ajouter des paires clé-valeur supplémentaires, par exemple lors de la journalisation des exceptions. Vous pouvez également fournir vos propres paramètres supplémentaires, comme décrit dans la section the section called "Paramètres de journal fournis par le client".

> **Note**
>
> Si votre code utilise déjà une autre bibliothèque de journalisation pour produire des journaux au format JSON, assurez-vous que le format de journal de votre fonction est défini sur du texte brut. Si vous définissez le format de journal sur JSON, les sorties de votre journal seront encodées deux fois.

L'exemple de commande de journalisation suivant montre comment écrire un message de journal de niveau `INFO`.

Example Code de journalisation .NET

```
context.Logger.LogInformation("Fetching cart from database");
```

Vous pouvez également utiliser une méthode de journal générique qui prend le niveau de journal comme argument, comme le montre l'exemple suivant.

```
context.Logger.Log(LogLevel.Information, "Fetching cart from database");
```

La sortie du journal par ces exemples d'extraits de code serait capturée dans CloudWatch Logs comme suit :

Example Enregistrement de journaux JSON

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
```

```
    "level": "Information",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "Fetching cart from database"
}
```

> **Note**
>
> Si vous configurez le format de journal de votre fonction pour utiliser du texte brut plutôt que du JSON, le niveau de journal capturé dans le message suit la convention de Microsoft qui consiste à utiliser une étiquette à quatre caractères. Par exemple, un niveau de journalisation `Debug` est représenté dans le message sous la forme `dbug`.
> Lorsque vous configurez votre fonction pour utiliser des journaux au format JSON, le niveau de journal capturé dans le journal utilise l'étiquette complète, comme indiqué dans l'exemple d'enregistrement de journal JSON.

Si vous n'attribuez aucun niveau à la sortie de votre journal, Lambda lui attribuera automatiquement le niveau INFO.

Exceptions de journalisation au format JSON

Lorsque vous utilisez la journalisation JSON structurée avec `ILambdaLogger`, vous pouvez journaliser les exceptions dans votre code, comme indiqué dans l'exemple suivant.

Example utilisation de la journalisation des exceptions

```
try
{
    connection.ExecuteQuery(query);
}
catch(Exception e)
{
    context.Logger.LogWarning(e, "Error executing query");
}
```

Le format de journal généré par ce code est illustré dans l'exemple de JSON suivant. Notez que la propriété `message` du JSON est renseignée à l'aide de l'argument message fourni dans l'appel `LogWarning`, tandis que la propriété `errorMessage` provient de la propriété `Message` de l'exception elle-même.

Example Enregistrement de journaux JSON

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Warning",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "Error executing query",
    "errorType": "System.Data.SqlClient.SqlException",
    "errorMessage": "Connection closed",
    "stackTrace": ["<call exception.StackTrace>"]
}
```

Si le format de journalisation de votre fonction est défini sur JSON, Lambda produit également des messages de journal au format JSON lorsque votre code génère une exception non détectée. L'exemple d'extrait de code et de message de journal suivants montre comment les exceptions non détectées sont journalisées.

Example code d'exception

```
throw new ApplicationException("Invalid data");
```

Example Enregistrement de journaux JSON

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Error",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "Invalid data",
    "errorType": "System.ApplicationException",
    "errorMessage": "Invalid data",
    "stackTrace": ["<call exception.StackTrace>"]
}
```

Paramètres de journal fournis par le client

Avec les messages de journal au format JSON, vous pouvez fournir des paramètres de journal supplémentaires et les inclure dans le journal `message`. L'exemple d'extrait de code suivant montre une commande permettant d'ajouter deux paramètres fournis par l'utilisateur et étiquetés

`retryAttempt` et `uri`. Dans l'exemple, la valeur de ces paramètres provient des arguments `retryAttempt` et `uriDestination` transmis à la commande de journalisation.

Example Commande de journalisation JSON avec paramètres supplémentaires

```
context.Logger.LogInformation("Starting retry {retryAttempt} to make GET request to
 {uri}", retryAttempt, uriDestination);
```

Le message du journal généré par cette commande est illustré dans l'exemple de fichier JSON suivant.

Example Enregistrement de journaux JSON

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Information",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "Starting retry 1 to make GET request to http://example.com/",
    "retryAttempt": 1,
    "uri": "http://example.com/"
}
```

Tip

Vous pouvez également utiliser des propriétés de positionnement plutôt que des noms lorsque vous spécifiez des paramètres supplémentaires. Par exemple, dans l'exemple précédent, la commande peut également être écrite comme suit :

```
context.Logger.LogInformation("Starting retry {0} to make GET request to {1}",
 retryAttempt, uriDestination);
```

Notez que lorsque vous fournissez des paramètres de journalisation supplémentaires, Lambda les capture en tant que propriétés de premier niveau dans l'enregistrement de journal JSON. Cette approche diffère de certaines bibliothèques de journalisation .NET populaires `Serilog`, telles que celles qui capturent des paramètres supplémentaires dans un objet enfant distinct.

Si l'argument que vous fournissez pour un paramètre supplémentaire est un objet complexe, Lambda utilise par défaut la méthode `ToString()` pour fournir la valeur. Pour indiquer qu'un argument doit

être sérialisé au format JSON, utilisez le préfixe @ tel qu'illustré dans l'extrait de code suivant. Dans cet exemple, `User` est un objet doté de propriétés `FirstName` et `LastName`.

Example Commande de journalisation JSON avec objet sérialisé JSON

```
context.Logger.LogInformation("User {@user} logged in", User);
```

Le message du journal généré par cette commande est illustré dans l'exemple de fichier JSON suivant.

Example Enregistrement de journaux JSON

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Information",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "User {@user} logged in",
    "user":
    {
        "FirstName": "John",
        "LastName": "Doe"
    }
}
```

Si l'argument d'un paramètre supplémentaire est un tableau ou implémente `IList` ou `IDictionary`, alors Lambda ajoute l'argument au message de journal JSON sous forme de tableau, comme indiqué dans l'exemple d'enregistrement de journal JSON suivant. Dans cet exemple, `{users}` prend un argument `IList` contenant des instances de la propriété `User` au même format que dans l'exemple précédent. Lambda le convertit cet `IList` en tableau, chaque valeur étant créée à l'aide de la méthode `ToString`.

Example Enregistrement de journal JSON avec un argument **`IList`**

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Information",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "{users} have joined the group",
```

```
    "users":
    [
        "Rosalez, Alejandro",
        "Stiles, John"
    ]
}
```

Vous pouvez également sérialiser la liste au format JSON en utilisant le préfixe `@` dans votre commande de journalisation. Dans l'exemple d'enregistrement de journal JSON suivant, la propriété `users` est sérialisée au format JSON.

Example Enregistrement de journal JSON avec un argument sérialisé JSON **`IList`**

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Information",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "{@users} have joined the group",
    "users":
    [
        {
            "FirstName": "Alejandro",
            "LastName": "Rosalez"
        },
        {
            "FirstName": "John",
            "LastName": "Stiles"
        }
    ]
}
```

Utilisation du filtrage au niveau du journal avec .NET

En configurant le filtrage au niveau des journaux, vous pouvez choisir de n'envoyer que les journaux d'un certain niveau de détail ou inférieur à CloudWatch Logs. Pour savoir comment configurer le filtrage de niveau journal de votre fonction, consultez the section called “Filtrage au niveau du journal”.

AWS Lambda Pour filtrer vos messages de journal par niveau de journal, vous pouvez soit utiliser des journaux au format JSON, soit utiliser les `Console` méthodes .NET pour générer des messages

de journal. Pour créer des journaux au format JSON, configurez le type de journal de votre fonction sur JSON et utilisez l'instance `ILambdaLogger`.

Avec les journaux formatés en JSON, Lambda filtre les sorties de votre journal à l'aide de la paire clé-valeur « niveau » de l'objet JSON décrit dans the section called "Utilisation du format de journal JSON structuré avec .NET".

Si vous utilisez les `Console` méthodes .NET pour écrire des messages dans CloudWatch Logs, Lambda applique les niveaux de journalisation à vos messages comme suit :

- Console. WriteLine méthode - Lambda applique un niveau de journalisation de `INFO`
- Méthode Console.Error : Lambda applique une journalisation de niveau `ERROR`

Lorsque vous configurez votre fonction pour utiliser le filtrage au niveau des journaux, vous devez sélectionner l'une des options suivantes pour le niveau de journaux que Lambda doit envoyer à Logs. CloudWatch Notez le mappage des niveaux de journalisation utilisés par Lambda avec les niveaux Microsoft standard utilisés par l'attribut .NET `ILambdaLogger`.

Niveau du journal de Lambda	Niveau Microsoft équivalent	Utilisation standard
TRACE (le plus détaillé)	Suivi	Les informations les plus précises utilisées pour tracer le chemin d'exécution de votre code
DEBUG	Débogage	Informations détaillées pour le débogage du système
INFO	Informations	Messages qui enregistrent le fonctionnement normal de votre fonction
WARN	Avertissement	Messages relatifs à des erreurs potentielles susceptib les d'entraîner un comportem ent inattendu si elles ne sont pas traitées

Niveau du journal de Lambda	Niveau Microsoft équivalent	Utilisation standard
ERROR	Erreur	Messages concernant les problèmes qui empêchent le code de fonctionner comme prévu
FATAL (moindre détail)	Critique	Messages relatifs à des erreurs graves entraînant l'arrêt du fonctionnement de l'application

Lambda envoie des journaux correspondant au niveau de détail sélectionné et inférieur à. CloudWatch Par exemple, si vous configurez un niveau de journalisation WARN, Lambda envoie des journaux correspondant aux niveaux WARN, ERROR et FATAL.

Outils de journalisation et bibliothèques supplémentaires

Powertools for AWS Lambda (.NET) est une boîte à outils destinée aux développeurs qui permet de mettre en œuvre les meilleures pratiques sans serveur et d'accroître la rapidité des développeurs. L'utilitaire Logging (français non garanti) fournit un enregistreur optimisé pour Lambda qui inclut des informations supplémentaires sur le contexte de fonction pour toutes vos fonctions avec une sortie structurée en JSON. Utilisez cet utilitaire pour effectuer les opérations suivantes :

- Capturer les champs clés du contexte Lambda, démarrer à froid et structurer la sortie de la journalisation sous forme de JSON
- Journaliser les événements d'invocation Lambda lorsque cela est demandé (désactivé par défaut)
- Imprimer tous les journaux uniquement pour un pourcentage d'invocations via l'échantillonnage des journaux (désactivé par défaut)
- Ajouter des clés supplémentaires au journal structuré à tout moment
- Utiliser un formateur de journaux personnalisé (Apportez votre propre formateur) pour produire des journaux dans une structure compatible avec le RFC de journalisation de votre organisation

Utilisation de Powertools pour AWS Lambda (.NET) et AWS SAM pour la journalisation structurée

Suivez les étapes ci-dessous pour télécharger, créer et déployer un exemple d'application Hello World C# avec des modules Powertools pour AWS Lambda (.NET) intégrés à l'aide du. AWS SAM Cette application met en œuvre un backend API de base et utilise Powertools pour émettre des journaux, des métriques et des traces. Elle se compose d'un point de terminaison Amazon API Gateway et d'une fonction Lambda. Lorsque vous envoyez une requête GET au point de terminaison API Gateway, la fonction Lambda appelle, envoie des journaux et des métriques au format métrique intégré à CloudWatch, et envoie des traces à. AWS X-Ray La fonction renvoie un message `hello world`.

Conditions préalables

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- .NET 8
- AWS CLI version 2
- AWS SAM CLI version 1.75 ou ultérieure. Si vous disposez d'une ancienne version de la AWS SAM CLI, reportez-vous à la section Mise à niveau de la AWS SAM CLI.

Déployer un exemple d' AWS SAM application

1. Initialisez l'application à l'aide du TypeScript modèle Hello World.

```
sam init --app-template hello-world-powertools-dotnet --name sam-app --package-type
 Zip --runtime dotnet6 --no-tracing
```

2. Créez l'application.

```
cd sam-app && sam build
```

3. Déployez l'application.

```
sam deploy --guided
```

4. Suivez les invites à l'écran. Appuyez sur `Enter` pour accepter les options par défaut fournies dans l'expérience interactive.

> **Note**
>
> Car l'autorisation n'a HelloWorldFunction peut-être pas été définie, est-ce que ça va ? , assurez-vous de participery.

5. Obtenez l'URL de l'application déployée :

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoquez le point de terminaison de l'API :

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

En cas de succès, vous obtiendrez cette réponse :

```
{"message":"hello world"}
```

7. Pour obtenir les journaux de la fonction, exécutez sam logs. Pour en savoir plus, consultez Utilisation des journaux dans le Guide du développeur AWS Serverless Application Model .

```
sam logs --stack-name sam-app
```

La sortie du journal se présente comme suit :

```
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8
 2025-09-20T14:15:27.988000 INIT_START Runtime Version:
 dotnet:6.v13        Runtime Version ARN: arn:aws:lambda:ap-
southeast-2::runtime:699f346a05dae24c58c45790bc4089f252bf17dae3997e79b17d939a288aa1ec
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2025-09-20T14:15:28.229000
 START RequestId: bed25b38-d012-42e7-ba28-f272535fb80e Version: $LATEST
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2025-09-20T14:15:29.259000
 2025-09-20T14:15:29.201Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
   {"_aws":{"Timestamp":1676902528962,"CloudWatchMetrics":[{"Namespace":"sam-
app-logging","Metrics":[{"Name":"ColdStart","Unit":"Count"}],"Dimensions":
[["FunctionName"],["Service"]]}]},"FunctionName":"sam-app-HelloWorldFunction-
haKIoVeose2p","Service":"PowertoolsHelloWorld","ColdStart":1}
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2025-09-20T14:15:30.479000
 2025-09-20T14:15:30.479Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
 {"ColdStart":true,"XrayTraceId":"1-63f3807f-5dbcb9910c96f50742707542","CorrelationId":"d3d
```

```
a549-4d67b2fdc015","FunctionName":"sam-app-HelloWorldFunction-
haKIoVeose2p","FunctionVersion":"$LATEST","FunctionMemorySize":256,"FunctionArn":"arn:aws:l
southeast-2:123456789012:function:sam-app-HelloWorldFunction-
haKIoVeose2p","FunctionRequestId":"bed25b38-d012-42e7-ba28-
f272535fb80e","Timestamp":"2025-09-20T14:15:30.4602970Z","Level":"Information","Service":"P
 world API - HTTP 200"}
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2025-09-20T14:15:30.599000
 2025-09-20T14:15:30.599Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
   {"_aws":{"Timestamp":1676902528922,"CloudWatchMetrics":[{"Namespace":"sam-
app-logging","Metrics":[{"Name":"ApiRequestCount","Unit":"Count"}],"Dimensions":
[["Service"]]}]},"Service":"PowertoolsHelloWorld","ApiRequestCount":1}
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2025-09-20T14:15:30.680000 END
 RequestId: bed25b38-d012-42e7-ba28-f272535fb80e
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2025-09-20T14:15:30.680000
 REPORT RequestId: bed25b38-d012-42e7-ba28-f272535fb80e  Duration: 2450.99 ms
  Billed Duration: 2692 ms Memory Size: 256 MB     Max Memory Used: 74 MB  Init
 Duration: 240.05 ms
XRAY TraceId: 1-63f3807f-5dbcb9910c96f50742707542       SegmentId: 16b362cd5f52cba0
```

8. Il s'agit d'un point de terminaison d'API public accessible par Internet. Nous vous recommandons de supprimer le point de terminaison après un test.

```
sam delete
```

Gestion de la conservation des journaux

Les groupes de journaux ne sont pas supprimés automatiquement quand vous supprimez une fonction. Pour éviter de stocker les journaux indéfiniment, supprimez le groupe de journaux ou configurez une période de conservation après laquelle les journaux CloudWatch sont automatiquement supprimés. Pour configurer la conservation des journaux, ajoutez ce qui suit à votre AWS SAM modèle :

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
```

```
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Affichage des journaux dans la console Lambda

Vous pouvez utiliser la console Lambda pour afficher la sortie du journal après avoir invoqué une fonction Lambda.

Si votre code peut être testé à partir de l'éditeur Code intégré, vous trouverez les journaux dans les résultats d'exécution. Lorsque vous utilisez la fonctionnalité de test de console pour invoquer une fonction, vous trouverez Sortie du journal dans la section Détails.

Afficher les journaux dans la CloudWatch console

Vous pouvez utiliser la CloudWatch console Amazon pour consulter les journaux de toutes les invocations de fonctions Lambda.

Pour afficher les journaux sur la CloudWatch console

1. Ouvrez la page Groupes de journaux sur la CloudWatch console.
2. Choisissez le groupe de journaux pour votre fonction (***your-function-name***/aws/lambda/).
3. Choisissez un flux de journaux.

Chaque flux de journal correspond à une instance de votre fonction. Un flux de journal apparaît lorsque vous mettez à jour votre fonction Lambda et lorsque des instances supplémentaires sont créées pour gérer les appels simultanés. Pour trouver les journaux d'un appel spécifique, nous vous recommandons d'instrumenter votre fonction avec. AWS X-Ray X-Ray enregistre des détails sur la demande et le flux de journaux dans le suivi.

Afficher les journaux à l'aide de AWS Command Line Interface (AWS CLI)

AWS CLI Il s'agit d'un outil open source qui vous permet d'interagir avec les AWS services à l'aide de commandes dans votre interface de ligne de commande. Pour effectuer les étapes de cette section, vous devez disposer de la version 2 de l'AWS CLI.

Vous pouvez utiliser AWS CLI pour récupérer les journaux d'une invocation à l'aide de l'option de commande `--log-type`. La réponse inclut un champ `LogResult` qui contient jusqu'à 4 Ko de journaux codés en base64 provenant de l'invocation.

Example récupérer un ID de journal

L'exemple suivant montre comment récupérer un ID de journal à partir du champ `LogResult` d'une fonction nommée `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Vous devriez voir la sortie suivante:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example décoder les journaux

Dans la même invite de commandes, utilisez l'utilitaire `base64` pour décoder les journaux. L'exemple suivant montre comment récupérer les journaux encodés en base64 pour `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

Vous devriez voir la sortie suivante :

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

L'utilitaire `base64` est disponible sous Linux, macOS et Ubuntu sous Windows. Les utilisateurs de macOS auront peut-être besoin d'utiliser `base64 -D`.

Example Script get-logs.sh

Dans la même invite de commandes, utilisez le script suivant pour télécharger les cinq derniers événements de journalisation. Le script utilise `sed` pour supprimer les guillemets du fichier de sortie et attend 15 secondes pour permettre la mise à disposition des journaux. La sortie comprend la réponse de Lambda, ainsi que la sortie de la commande `get-log-events`.

Copiez le contenu de l'exemple de code suivant et enregistrez-le dans votre répertoire de projet Lambda sous `get-logs.sh`.

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS et Linux (uniquement)

Dans la même invite de commandes, les utilisateurs macOS et Linux peuvent avoir besoin d'exécuter la commande suivante pour s'assurer que le script est exécutable.

```
chmod -R 755 get-logs.sh
```

Example récupérer les cinq derniers événements de journal

Dans la même invite de commande, exécutez le script suivant pour obtenir les cinq derniers événements de journalisation.

```
./get-logs.sh
```

Vous devriez voir la sortie suivante:

```
{
    "StatusCode": 200,
```

```
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

Suppression de journaux

Les groupes de journaux ne sont pas supprimés automatiquement quand vous supprimez une fonction. Pour éviter de stocker des journaux indéfiniment, supprimez le groupe de journaux ou configurez une période de conservation à l'issue de laquelle les journaux sont supprimés automatiquement.

Instrumentation du code C # dans AWS Lambda

Lambda s'intègre avec AWS X-Ray pour vous aider à suivre, déboguer et optimiser les applications Lambda. Vous pouvez utiliser X-Ray pour suivre une demande lorsque celle-ci parcourt les ressources de votre application, qui peuvent inclure des fonctions Lambda et d'autres services AWS.

Pour envoyer des données de traçage à X-Ray, vous pouvez utiliser l'une des trois bibliothèques SDK suivantes :

- AWS Distro de OpenTelemetry (ADOT) : solution sécurisée, prête pour la production, distribution prise en charge par AWS du kit SDK OpenTelemetry (OTel).
- Kit SDK AWS X-Ray pour .NET – Un kit SDK permettant de générer et d'envoyer des données de suivi à X-Ray.
- Powertools pour AWS Lambda (.NET) : une boîte à outils pour développeurs permettant de mettre en œuvre les bonnes pratiques en matière de sans serveur et d'augmenter la rapidité des développeurs.

Chacun des kits SDK offre des moyens d'envoyer vos données de télémétrie au service X-Ray. Vous pouvez ensuite utiliser X-Ray pour afficher, filtrer et avoir un aperçu des métriques de performance de votre application, afin d'identifier les problèmes et les occasions d'optimiser votre application.

> **Important**
>
> X-Ray et Powertools pour les kits SDK AWS Lambda font partie d'une solution d'instrumentation étroitement intégrée proposée par AWS. Les couches ADOT Lambda font partie d'une norme industrielle pour l'instrumentation de traçage qui collecte plus de données en général, mais qui peut ne pas convenir à tous les cas d'utilisation. Vous pouvez mettre en œuvre le traçage de bout en bout dans X-Ray en utilisant l'une ou l'autre solution. Pour en savoir plus sur le choix entre les deux, consultez Choix entre les kits SDK AWS Distro for OpenTelemetry et X-Ray (français non garanti).

Sections

- Utilisation de Powertools pour AWS Lambda (.NET) et de AWS SAM pour le traçage
- Utilisation du kit SDK X-Ray pour instrumenter vos fonctions .NET
- Activation du suivi avec la console Lambda

- Activation du suivi avec l'API Lambda
- Activation du suivi avec CloudFormation
- Interprétation d'un suivi X-Ray

Utilisation de Powertools pour AWS Lambda (.NET) et de AWS SAM pour le traçage

Suivez les étapes ci-dessous pour télécharger, créer et déployer un exemple d'application Hello World C# avec des modules Powertools pour AWS Lambda (.NET) intégrés à l'aide de AWS SAM. Cette application met en œuvre un backend API de base et utilise Powertools pour émettre des journaux, des métriques et des traces. Elle se compose d'un point de terminaison Amazon API Gateway et d'une fonction Lambda. Lorsque vous envoyez une demande GET au point de terminaison API Gateway, la fonction Lambda est invoquée, envoie des journaux et des métriques à l'aide du format Embedded Metric à CloudWatch, et envoie des traces à AWS X-Ray. La fonction renvoie un message hello world.

Prérequis

Pour exécuter la procédure indiquée dans cette section, vous devez satisfaire aux exigences suivantes :

- .NET 8
- AWS CLI Version 2 de 2
- AWS SAM CLI version 1.75 ou ultérieure. Si vous avez une version antérieure de la CLI AWS SAM, consultez Mise à niveau de la CLI AWS SAM.

Déployer un exemple d'application AWS SAM

1. Initialisez l'application à l'aide du modèle Hello World TypeScript.

```
sam init --app-template hello-world-powertools-dotnet --name sam-app --package-type
 Zip --runtime dotnet6 --no-tracing
```

2. Créez l'application.

```
cd sam-app && sam build
```

3. Déployez l'application.

```
sam deploy --guided
```

4. Suivez les invites à l'écran. Appuyez sur `Enter` pour accepter les options par défaut fournies dans l'expérience interactive.

> **Note**
>
> Pour HelloWorldFunction n'a peut-être pas d'autorisation définie, est-ce correct ?, assurez-vous de saisir y.

5. Obtenez l'URL de l'application déployée :

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoquez le point de terminaison de l'API :

```
curl <URL_FROM_PREVIOUS_STEP>
```

En cas de succès, vous obtiendrez cette réponse :

```
{"message":"hello world"}
```

7. Pour obtenir les traces de la fonction, exécutez sam traces.

```
sam traces
```

La sortie de la trace ressemble à ceci :

```
New XRay Service Graph
  Start time: 2023-02-20 23:05:16+08:00
  End time: 2023-02-20 23:05:16+08:00
  Reference Id: 0 - AWS::Lambda - sam-app-HelloWorldFunction-pNjujb7mEoew - Edges:
 [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
```

```
    - fault count(5XX): 0
    - total response time: 2.814
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-pNjujb7mEoew
 - Edges: []
   Summary_statistics:
    - total requests: 1
    - ok count(2XX): 1
    - error count(4XX): 0
    - fault count(5XX): 0
    - total response time: 2.429
  Reference Id: 2 - (Root) AWS::ApiGateway::Stage - sam-app/Prod - Edges: [0]
   Summary_statistics:
    - total requests: 1
    - ok count(2XX): 1
    - error count(4XX): 0
    - fault count(5XX): 0
    - total response time: 2.839
  Reference Id: 3 - client - sam-app/Prod - Edges: [2]
   Summary_statistics:
    - total requests: 0
    - ok count(2XX): 0
    - error count(4XX): 0
    - fault count(5XX): 0
    - total response time: 0

XRay Event [revision 3] at (2023-02-20T23:05:16.521000) with id
 (1-63f38c2c-270200bf1d292a442c8e8a00) and duration (2.877s)
 - 2.839s - sam-app/Prod [HTTP: 200]
   - 2.836s - Lambda [HTTP: 200]
 - 2.814s - sam-app-HelloWorldFunction-pNjujb7mEoew [HTTP: 200]
 - 2.429s - sam-app-HelloWorldFunction-pNjujb7mEoew
   - 0.230s - Initialization
   - 2.389s - Invocation
     - 0.600s - ## FunctionHandler
       - 0.517s - Get Calling IP
   - 0.039s - Overhead
```

8. Il s’agit d’un point de terminaison d’API public accessible par Internet. Nous vous recommandons de supprimer le point de terminaison après un test.

```
sam delete
```

X-Ray ne trace pas toutes les requêtes vers votre application. X-Ray applique un algorithme d'échantillonnage pour s'assurer que le suivi est efficace, tout en fournissant un échantillon représentatif de toutes les demandes. Le taux d'échantillonnage est 1 demande par seconde et 5 % de demandes supplémentaires. Vous ne pouvez pas configurer ce taux d'échantillonnage X-Ray pour vos fonctions.

Utilisation du kit SDK X-Ray pour instrumenter vos fonctions .NET

Vous pouvez instrumenter le code de gestionnaire pour enregistrer les métadonnées et suivre les appels en aval. Pour enregistrer les détails des appels que votre fonction effectue vers d'autres ressources et services, utilisez le Kit SDK AWS X-Ray pour .NET. Pour obtenir ce kit SDK, ajoutez les packages `AWSXRayRecorder` à votre fichier de projet.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <GenerateRuntimeConfigurationFiles>true</GenerateRuntimeConfigurationFiles>
    <AWSProjectType>Lambda</AWSProjectType>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Amazon.Lambda.Core" Version="2.1.0" />
    <PackageReference Include="Amazon.Lambda.SQSEvents" Version="2.1.0" />
    <PackageReference Include="Amazon.Lambda.Serialization.Json" Version="2.1.0" />
    <PackageReference Include="AWSSDK.Core" Version="3.7.103.24" />
    <PackageReference Include="AWSSDK.Lambda" Version="3.7.104.3" />
    <PackageReference Include="AWSXRayRecorder.Core" Version="2.13.0" />
    <PackageReference Include="AWSXRayRecorder.Handlers.AwsSdk" Version="2.11.0" />
  </ItemGroup>
</Project>
```

Il existe une série de packages Nuget qui fournissent une auto-instrumentation pour les kits SDK AWS, Entity Framework et les requêtes HTTP. Pour connaître l'ensemble des options de configuration, reportez-vous à kit SDK AWS X-Ray pour .NET dans le Guide du développeur AWS X-Ray.

Une fois que vous avez ajouté les packages Nuget de votre choix, configurez l'auto-instrumentation. La meilleure pratique consiste à effectuer cette configuration en dehors de la fonction du gestionnaire de votre fonction. Cela vous permet de tirer parti de la réutilisation de l'environnement d'exécution afin d'améliorer les performances de votre fonction. Dans l'exemple de code suivant, la méthode

`RegisterXRayForAllServices` est appelée dans le constructeur de la fonction pour ajouter l'instrumentation pour tous les appels du kit SDK AWS.

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function()
    {
        // Add auto instrumentation for all AWS SDK calls
        // It is important to call this method before initializing any SDK clients
        AWSSDKHandler.RegisterXRayForAllServices();
        this._repo = new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request)
    {
        var id = request.PathParameters["id"];

        var databaseRecord = await this._repo.GetById(id);

        return new APIGatewayProxyResponse
        {
            StatusCode = (int)HttpStatusCode.OK,
            Body = JsonSerializer.Serialize(databaseRecord)
        };
    }
}
```

Activation du suivi avec la console Lambda

Pour activer/désactiver le traçage actif sur votre fonction Lambda avec la console, procédez comme suit :

Pour activer le traçage actif

1. Ouvrez la page Functions (Fonctions) de la console Lambda.

2. Choisissez une fonction.
3. Choisissez Configuration, puis choisissez Outils de surveillance et d'opérations.
4. Dans Outils de surveillance supplémentaires, choisissez Modifier.
5. Sous CloudWatch Application Signals et AWS X-Ray sélectionnez Activer pour Suivis de service Lambda.
6. Choisissez Enregistrer.

Activation du suivi avec l'API Lambda

Configurez le suivi sur votre fonction Lambda avec le AWS CLI ou le kit SDK AWS, utilisez les opérations d'API suivantes :

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

L'exemple de commande AWS CLI suivant active le suivi actif sur une fonction nommée my-function.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Le mode de suivi fait partie de la configuration spécifique de la version lorsque vous publiez une version de votre fonction. Vous ne pouvez pas modifier le mode de suivi sur une version publiée.

Activation du suivi avec CloudFormation

Pour activer le suivi d'une ressource `AWS::Lambda::Function` dans un modèle CloudFormation, utilisez la propriété `TracingConfig`.

Example function-inline.yml – Configuration du suivi

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
```

```
      ...
```

Pour une ressource AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function`, utilisez la propriété `Tracing`.

Example template.yml – Configuration du suivi

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

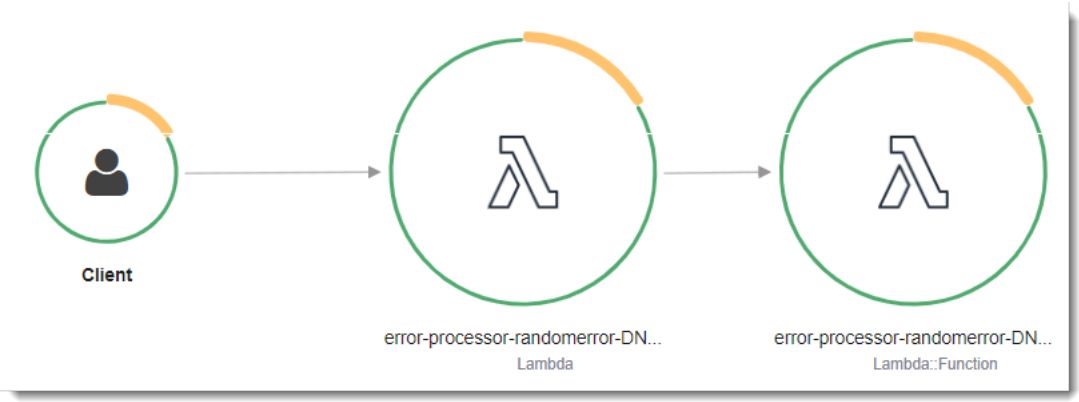
Interprétation d'un suivi X-Ray

Votre fonction a besoin d'une autorisation pour charger des données de suivi vers X-Ray. Lorsque vous activez le suivi actif dans la console Lambda, Lambda ajoute les autorisations requises au rôle d'exécution de votre fonction. Sinon, ajoutez la politique AWSXRayDaemonWriteAccess au rôle d'exécution.

Une fois que vous avez configuré le suivi actif, vous pouvez observer des demandes spécifiques via votre application. Le graphique de services X-Ray affiche des informations sur votre application et tous ses composants. L'exemple suivant montre une application dotée de deux fonctions. La fonction principale traite les événements et renvoie parfois des erreurs. La deuxième fonction en partant du haut traite les erreurs qui apparaissent dans le groupe de journaux du premier et utilise le kit SDK AWS pour appeler X-Ray, Amazon Simple Storage Service (Amazon S3) et Amazon CloudWatch logs.

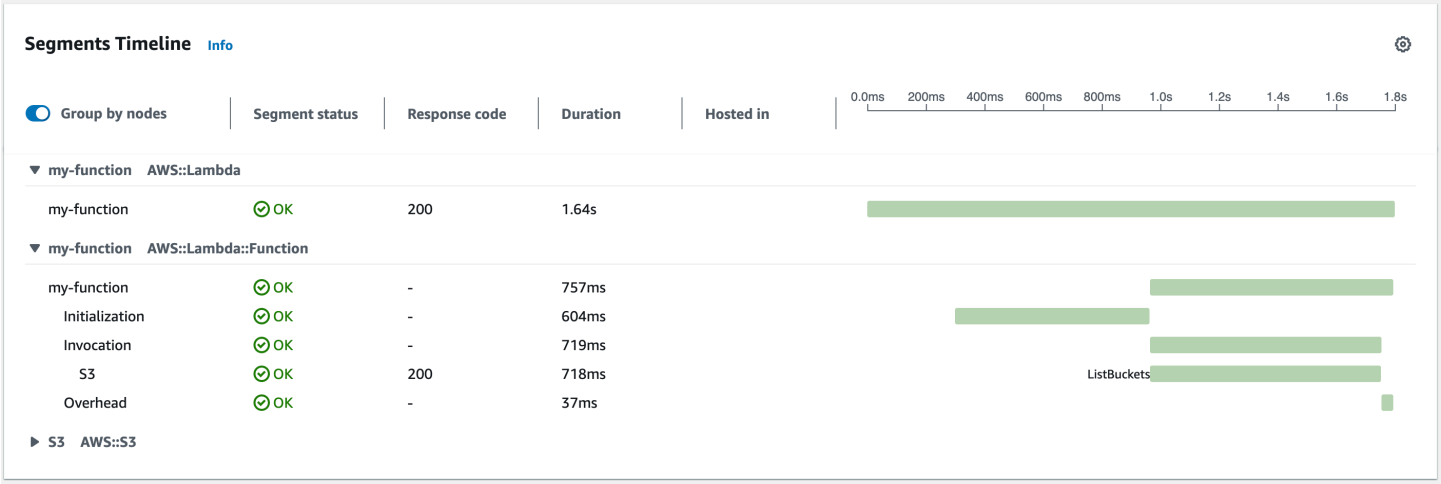
X-Ray ne trace pas toutes les requêtes vers votre application. X-Ray applique un algorithme d'échantillonnage pour s'assurer que le suivi est efficace, tout en fournissant un échantillon représentatif de toutes les demandes. Le taux d'échantillonnage est 1 demande par seconde et 5 % de demandes supplémentaires. Vous ne pouvez pas configurer ce taux d'échantillonnage X-Ray pour vos fonctions.

Dans X-Ray, un suivi enregistre des informations sur une demande traitée par un ou plusieurs services. Lambda enregistre deux segments par suivi, ce qui a pour effet de créer deux nœuds sur le graphique du service. L'image suivante met en évidence ces deux nœuds :



Le premier nœud sur la gauche représente le service Lambda qui reçoit la demande d'invocation. Le deuxième nœud représente votre fonction Lambda spécifique. L'exemple suivant illustre une trace avec ces deux segments. Les deux sont nommés my-function, mais l'un a pour origine `AWS::Lambda` et l'autre a pour origine `AWS::Lambda::Function`. Si le segment `AWS::Lambda` affiche une erreur, cela signifie que le service Lambda a rencontré un problème. Si le segment

`AWS::Lambda::Function` affiche une erreur, cela signifie que votre fonction a rencontré un problème.



Cet exemple développe le segment `AWS::Lambda::Function` pour afficher ses trois sous-segments.

> **Note**
>
> AWS est en train de modifier le service Lambda. En raison de ces modifications, vous pouvez constater des différences mineures entre la structure et le contenu des messages du journal système et des segments de suivi émis par les différentes fonctions Lambda de votre Compte AWS.
> L'exemple de suivi présenté ici illustre le segment de fonction à l'ancienne. Les différences entre les segments à l'ancienne et de style moderne sont décrites dans les paragraphes suivants.
> Ces modifications seront mises en œuvre au cours des prochaines semaines, et toutes les fonctions dans toutes les Régions AWS, sauf les régions de Chine et GovCloud, passeront à l'utilisation du nouveau format des messages de journal et des segments de suivi.

Le segment de fonction à l'ancienne contient les sous-segments suivants :

- Initialization (Initialisation) : représente le temps passé à charger votre fonction et à exécuter le code d'initialisation. Ce sous-segment apparaît pour le premier événement traité par chaque instance de votre fonction.
- Invocation – Représente le temps passé à exécuter votre code de gestionnaire.

- Overhead (Travail supplémentaire) – Représente le temps que le fichier d'exécution Lambda passe à se préparer à gérer l'événement suivant.

Le segment de fonction de style moderne ne contient pas de sous-segment `Invocation`. À la place, les sous-segments du client sont directement rattachés au segment de fonction. Pour plus d'informations sur la structure des segments de fonction à l'ancienne et de style moderne, consultez the section called "Comprendre les suivis X-Ray".

Vous pouvez également utiliser des clients HTTP, enregistrer des requêtes SQL et créer des sous-segments personnalisés avec des annotations et des métadonnées. Pour plus d'informations, consultez Kit SDK AWS X-Ray pour .NET dans le AWS X-Ray Guide du développeur.

> **Tarification**
>
> Vous pouvez utiliser gratuitement le traçage par X-Ray chaque mois jusqu'à une certaine limite dans le cadre de l'offre gratuite AWS. Au-delà de ce seuil, X-Ray facture le stockage et la récupération du suivi. Pour en savoir plus, consultez Tarification de AWS X-Ray.

Test de fonction AWS Lambda en C#

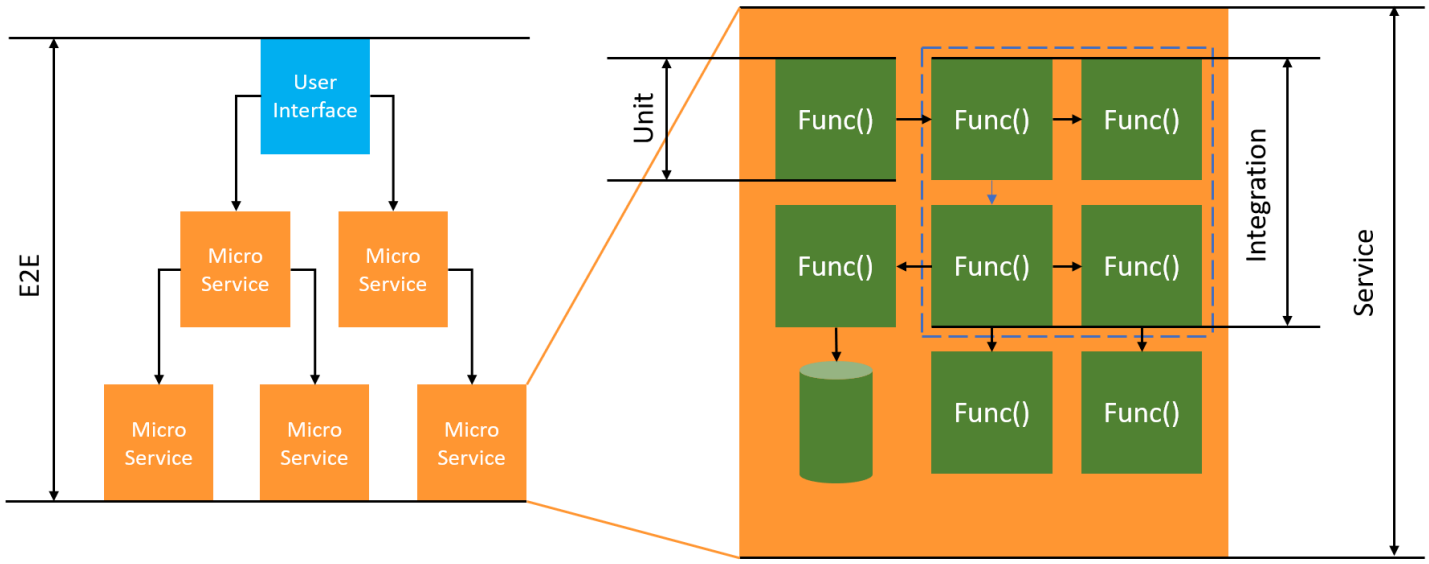
> **Note**
>
> Consultez le chapitre Test des fonctions pour une présentation complète des techniques et des bonnes pratiques pour tester les solutions sans serveur.

Le test des fonctions sans serveur utilise les types et techniques de tests traditionnels, mais vous devez également envisager de tester les applications sans serveur dans leur ensemble. Les tests basés sur le cloud fourniront la mesure la plus précise de la qualité de vos fonctions et de vos applications sans serveur.

Une architecture d'application sans serveur comprend des services gérés qui fournissent des fonctionnalités d'applications critiques par le biais d'appels d'API. C'est pourquoi votre cycle de développement doit inclure des tests automatisés qui vérifient la fonctionnalité lorsque votre fonction et vos services interagissent.

Si vous ne créez pas de tests basés sur le cloud, vous pouvez rencontrer des problèmes en raison des différences entre votre environnement local et l'environnement déployé. Votre processus d'intégration continue doit exécuter des tests sur une série de ressources allouées dans le cloud avant de promouvoir votre code vers l'environnement de déploiement suivant, comme l'assurance qualité, la mise en place ou la production.

Poursuivez la lecture de ce petit guide pour en savoir plus sur les stratégies de test pour les applications sans serveur, ou rendez-vous sur le référentiel Serverless Test Samples pour vous plonger dans des exemples pratiques, spécifiques au langage et à l'exécution que vous avez choisis.



Pour les tests sans serveur, vous écrirez toujours des tests unitaires, d'intégration et de bout en bout.

- Tests unitaires : tests exécutés sur un bloc de code isolé. Par exemple, la vérification de la logique métier permettant de calculer les frais de livraison en fonction d'un article et d'une destination donnés.
- Tests d'intégration : tests impliquant au moins deux composants ou services qui interagissent, généralement dans un environnement cloud. Par exemple, la vérification d'une fonction traite les événements d'une file d'attente.
- Tests de bout en bout : tests qui vérifient le comportement de l'ensemble d'une application. Il s'agit par exemple de s'assurer que l'infrastructure est correctement mise en place et que les événements circulent entre les services comme prévu pour enregistrer la commande d'un client.

Test de vos applications sans serveur

Vous utiliserez généralement une combinaison d'approches pour tester le code de votre application sans serveur, notamment des tests dans le cloud, des tests avec des simulations et, occasionnellement, des tests avec des émulateurs.

Tests dans le cloud

Les tests dans le cloud sont utiles pour toutes les phases des tests, y compris les tests unitaires, les tests d'intégration et les tests de bout en bout. Vous exécutez des tests sur du code déployé dans le cloud et interagissez avec des services basés sur le cloud. Cette approche fournit la mesure la plus précise de la qualité de votre code.

Un moyen pratique de déboguer votre fonction Lambda dans le cloud est de passer par la console avec un événement de test. Un événement de test est une entrée JSON pour votre fonction. Si votre fonction ne nécessite pas d'entrée, l'événement peut être un document JSON vide (`{}`). La console fournit des exemples d'événements pour diverses intégrations de services. Après avoir créé un événement dans la console, vous pouvez le partager avec votre équipe pour faciliter les tests et les rendre plus cohérents.

> **Note**
>
> Tester une fonction dans la console est un moyen rapide de démarrer, mais l'automatisation de vos cycles de test garantit la qualité des applications et la rapidité du développement.

Outils de test

Pour accélérer votre cycle de développement, il existe un certain nombre d'outils et de techniques que vous pouvez utiliser pour tester vos fonctions. Par exemple, Accélérer AWS SAM et le mode de surveillance AWS CDK réduisent tous deux le temps nécessaire pour mettre à jour les environnements cloud.

La façon dont vous définissez votre code de fonction Lambda facilite l'ajout de tests d'unité. Lambda nécessite un constructeur public sans paramètre pour initialiser votre catégorie. L'introduction d'un deuxième constructeur interne vous permet de contrôler les dépendances utilisées par votre application.

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function(): this(null)
    {
    }

    internal Function(IDatabaseRepository repo)
    {
```

```
        this._repo = repo ?? new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request)
    {
        var id = request.PathParameters["id"];

        var databaseRecord = await this._repo.GetById(id);

        return new APIGatewayProxyResponse
        {
            StatusCode = (int)HttpStatusCode.OK,
            Body = JsonSerializer.Serialize(databaseRecord)
        };
    }
}
```

Pour écrire un test pour cette fonction, vous pouvez initialiser une nouvelle instance de votre catégorie `Function` et transmettre une implémentation simulée du `IDatabaseRepository`. Les exemples ci-dessous utilisent `XUnit`, `Moq`, et `FluentAssertions` pour écrire un test simple permettant de s'assurer que le `FunctionHandler` renvoie un code d'état 200.

```
using Xunit;
using Moq;
using FluentAssertions;

public class FunctionTests
{
    [Fact]
    public async Task TestLambdaHandler_WhenInputIsValid_ShouldReturn200StatusCode()
    {
        // Arrange
        var mockDatabaseRepository = new Mock<IDatabaseRepository>();

        var functionUnderTest = new Function(mockDatabaseRepository.Object);

        // Act
        var response = await functionUnderTest.FunctionHandler(new
 APIGatewayProxyRequest());

        // Assert
        response.StatusCode.Should().Be(200);
```

```
    }
}
```

Pour obtenir des exemples plus détaillés, y compris des exemples de tests asynchrones, consultez le référentiel d'échantillons de tests .NET dans GitHub.

Création de fonctions Lambda avec PowerShell

Les sections suivantes expliquent comment les modèles de programmation courants et les concepts de base s'appliquent lorsque vous créez du code de fonction Lambda dans. PowerShell

Lambda fournit les exemples d'applications suivants pour : PowerShell

- blank-powershell — PowerShell Fonction qui montre l'utilisation de la journalisation, des variables d'environnement et du SDK. AWS

Avant de commencer, vous devez d'abord configurer un environnement PowerShell de développement. Pour obtenir des instructions sur la façon de procéder, veuillez consulter Configuration d'un environnement de développement PowerShell.

Pour savoir comment utiliser le AWSLambda PSCore module pour télécharger des exemples de PowerShell projets à partir de modèles, créer des packages de PowerShell déploiement et déployer PowerShell des fonctions AWS dans le cloud, consultezDéployer des fonctions Lambda PowerShell avec des archives de fichiers .zip.

Lambda fournit les exécutions suivantes pour les langages .NET :

Name	Identifiant	Système d'exploitation	Date d'obsolescence	Créer la fonction de blocage	Mettre à jour la fonction de blocage
.NET 10	`dotnet10`	Amazon Linux 2	14 novembre 2028	14 déc. 2028	15 janvier 2029
.NET 9 (conteneur uniquement)	`dotnet9`	Amazon Linux 2	10 novembre 2026	Non planifié	Non planifié
.NET 8	`dotnet8`	Amazon Linux 2	10 novembre 2026	10 déc. 2026	11 janvier 2027

Rubriques

- Configuration d'un environnement de développement PowerShell

- Déployer des fonctions Lambda PowerShell avec des archives de fichiers .zip
- Définition du gestionnaire de fonction Lambda dans PowerShell
- Utilisation de l’objet de contexte Lambda pour récupérer les informations de la fonction PowerShell
- Journalisation et surveillance des fonctions Lambda Powershell

Configuration d’un environnement de développement PowerShell

Lambda fournit un ensemble d’outils et de bibliothèques pour l’environnement d’exécution PowerShell. Pour obtenir des instructions d’installation, consultez Outils Lambda pour PowerShell sur GitHub.

Le module AWSLambdaPSCore inclut les cmdlets suivantes pour vous aider à créer et publier des fonctions Lambda PowerShell.

- Get-AWSPowerShellLambdaTemplate – Renvoie une liste de modèles de mise en route.
- New-AWSPowerShellLambda – Crée un script PowerShell basé sur un modèle.
- Publish-AWSPowerShellLambda – Publie un script PowerShell donné pour Lambda.
- New-AWSPowerShellLambdaPackage – Crée un package de déploiement Lambda que vous pouvez utiliser dans un système de CI/CD pour le déploiement.

Déployer des fonctions Lambda PowerShell avec des archives de fichiers .zip

Un package de déploiement pour le runtime PowerShell contient votre script PowerShell, les modules PowerShell requis pour votre script PowerShell, ainsi que les assemblies nécessaires pour héberger PowerShell Core.

Création de la fonction Lambda

Pour commencer à écrire et appeler un script PowerShell avec Lambda, vous pouvez utiliser la cmdlet `New-AWSPowerShellLambda` pour créer un script de démarrage basé sur un modèle. Vous pouvez utiliser la cmdlet `Publish-AWSPowerShellLambda` pour déployer votre script vers Lambda. Ensuite, vous pouvez tester votre script via la ligne de commande ou la console Lambda.

Pour créer un nouveau script PowerShell, le charger et le tester, procédez aux étapes suivantes :

1. Pour afficher la liste des modèles disponibles, exécutez la commande suivante :

   ```
   PS C:\> Get-AWSPowerShellLambdaTemplate

   Template               Description
   --------               -----------
   Basic                  Bare bones script
   CodeCommitTrigger      Script to process AWS CodeCommit Triggers
   ...
   ```

2. Pour créer un exemple de script basé sur le modèle `Basic`, exécutez la commande suivante :

   ```
   New-AWSPowerShellLambda -ScriptName MyFirstPSScript -Template Basic
   ```

   Un nouveau fichier nommé `MyFirstPSScript.ps1` est créé dans un nouveau sous-répertoire du répertoire actuel. Le nom de ce répertoire est basé sur le paramètre `-ScriptName`. Vous pouvez utiliser le paramètre `-Directory` pour choisir un autre répertoire.

   Vous pouvez voir que le nouveau fichier a le contenu suivant :

   ```
   # PowerShell script file to run as a Lambda function
   #
   # When executing in Lambda the following variables are predefined.
   ```

```
#   $LambdaInput - A PSObject that contains the Lambda function input data.
#   $LambdaContext - An Amazon.Lambda.Core.ILambdaContext object that contains
 information about the currently running Lambda environment.
#
# The last item in the PowerShell pipeline is returned as the result of the Lambda
 function.
#
# To include PowerShell modules with your Lambda function, like the
 AWSPowerShell.NetCore module, add a "#Requires" statement
# indicating the module and version.

#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}

# Uncomment to send the input to CloudWatch Logs
# Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 5)
```

3. Pour voir comment les messages du journal de votre script PowerShell sont envoyés à Amazon CloudWatch Logs, supprimez le commentaire de la ligne `Write-Host` de l'exemple de script.

   Pour illustrer comment renvoyer des données à partir de vos fonctions Lambda, ajoutez une ligne à la fin du script avec `$PSVersionTable`. Cela ajoute `$PSVersionTable` dans le pipeline PowerShell. Une fois l'exécution du script PowerShell terminée, le dernier objet figurant dans le pipeline PowerShell correspond aux données de retour de la fonction Lambda. `$PSVersionTable` est une variable globale PowerShell qui fournit également des informations sur l'environnement d'exécution.

   Après avoir effectué ces modifications, les deux dernières lignes de l'exemple de script ressemblent à ce qui suit :

   ```
   Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 5)
   $PSVersionTable
   ```

4. Après avoir modifié le fichier `MyFirstPSScript.ps1`, modifiez le répertoire en spécifiant l'emplacement du script. Exécutez ensuite la commande suivante pour publier le script dans Lambda :

   ```
   Publish-AWSPowerShellLambda -ScriptPath .\MyFirstPSScript.ps1 -Name
    MyFirstPSScript -Region us-east-2
   ```

   Notez que le paramètre `-Name` spécifie le nom de la fonction Lambda qui s'affiche dans la console Lambda. Vous pouvez utiliser cette fonction pour appeler manuellement votre script.

5. Appelez votre fonction à l'aide de la commande AWS Command Line Interface (AWS CLI) `invoke`.

```
> aws lambda invoke --function-name MyFirstPSScript out
```

Définition du gestionnaire de fonction Lambda dans PowerShell

Lors de l'appel d'une fonction Lambda, le gestionnaire Lambda appelle le script PowerShell.

Lorsque le script PowerShell est appelé, les variables suivantes sont prédéfinies :

- *$LambdaInput* – Objet PSObject contenant l'entrée pour le gestionnaire. Ces données d'entrée peuvent être des données d'événement (publiées par une source d'événement) ou des données d'entrée personnalisées que vous fournissez, telles qu'une chaîne ou n'importe quel objet de données personnalisé.
- *$LambdaContext* – Objet Amazon.Lambda.Core.ILambdaContext qui vous permet d'accéder à des informations sur l'appel en cours, comme le nom de la fonction en cours d'exécution, la limite de mémoire, le temps d'exécution restant et la journalisation.

Prenons l'exemple de code PowerShell suivant.

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host 'Function Name:' $LambdaContext.FunctionName
```

Ce script renvoie la propriété FunctionName qui est obtenue à partir de la variable $LambdaContext.

> **Note**
>
> Vous êtes tenu d'utiliser l'instruction `#Requires` dans vos scripts PowerShell pour indiquer les modules dont vos scripts dépendent. Cette instruction effectue deux tâches importantes. 1) Elle communique aux autres développeurs les modules que le script utilise, et 2) elle identifie les modules dépendants que les outils AWS PowerShell doivent mettre en package avec le script, dans le cadre du déploiement. Pour plus d'informations sur l'instruction `#Requires` dans PowerShell, consultez About Requires (À propos de Requires). Pour de plus amples informations sur les packages de déploiement PowerShell, veuillez consulter Déployer des fonctions Lambda PowerShell avec des archives de fichiers .zip.
> Lorsque votre fonction Lambda PowerShell utilise les cmdlets AWS PowerShell, veillez à définir une instruction `#Requires` faisant référence au module `AWSPowerShell.NetCore` qui prend en charge PowerShell Core, et non au module `AWSPowerShell` qui prend en charge uniquement Windows PowerShell. De plus, veillez à utiliser la version 3.3.270.0 ou plus récente de `AWSPowerShell.NetCore`, qui optimise le processus d'importation d'applet

de commande. Si vous utilisez une version plus ancienne, vous connaîtrez des démarrages à froid plus longs. Pour plus d'informations, consultez Tools for PowerShell AWS.

Renvoi de données

Certains appels Lambda sont destinés à renvoyer des données à leur appelant. Par exemple, si un appel répond à une demande web provenant d'API Gateway, notre fonction Lambda doit renvoyer la réponse. Pour PowerShell Lambda, le dernier objet ajouté dans le pipeline PowerShell correspond aux données de retour de l'appel Lambda. Si l'objet est une chaîne, les données sont renvoyées en l'état. Dans le cas contraire, l'objet est converti en données JSON à l'aide de l'applet de commande `ConvertTo-Json`.

Par exemple, prenez en considération l'instruction PowerShell suivante, qui ajoute `$PSVersionTable` dans le pipeline PowerShell :

```
$PSVersionTable
```

Une fois le script PowerShell terminé, le dernier objet figurant dans le pipeline PowerShell correspond aux données de retour de la fonction Lambda. `$PSVersionTable` est une variable globale PowerShell qui fournit également des informations sur l'environnement d'exécution.

Utilisation de l’objet de contexte Lambda pour récupérer les informations de la fonction PowerShell

Lorsque Lambda exécute votre fonction, il transmet les informations de contexte en mettant une variable `$LambdaContext` à la disposition du gestionnaire. Cette variable fournit des méthodes et des propriétés avec des informations sur l'appel, la fonction et l'environnement d'exécution.

Propriétés du contexte

- `FunctionName` – Nom de la fonction Lambda.
- `FunctionVersion` – Version de la fonction.
- `InvokedFunctionArn` – Amazon Resource Name (ARN) utilisé pour appeler la fonction. Indique si l’appelant a spécifié un numéro de version ou un alias.
- `MemoryLimitInMB` – Quantité de mémoire allouée à la fonction.
- `AwsRequestId` – Identifiant de la demande d’invocation.
- `LogGroupName` – Groupe de journaux pour la fonction.
- `LogStreamName` – Flux de journal de l'instance de fonction.
- `RemainingTime` – Nombre de millisecondes restant avant l'expiration de l'exécution.
- `Identity` – (applications mobiles) Informations sur l'identité Amazon Cognito qui a autorisé la demande.
- `ClientContext` – (applications mobiles) Contexte client fourni à Lambda par l'application client.
- `Logger` – Objet enregistreur pour la fonction.

L'extrait de code PowerShell suivant illustre une fonction simple de gestionnaire qui affiche certaines informations de contexte.

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host 'Function name:' $LambdaContext.FunctionName
Write-Host 'Remaining milliseconds:' $LambdaContext.RemainingTime.TotalMilliseconds
Write-Host 'Log group name:' $LambdaContext.LogGroupName
Write-Host 'Log stream name:' $LambdaContext.LogStreamName
```

Journalisation et surveillance des fonctions Lambda Powershell

AWS Lambda surveille automatiquement les fonctions Lambda en votre nom et envoie les journaux à Amazon. CloudWatch Votre fonction Lambda est fournie avec un groupe de CloudWatch journaux Logs et un flux de journaux pour chaque instance de votre fonction. L'environnement d'exécution Lambda envoie des informations sur chaque invocation au flux de journaux et transmet les journaux et autres sorties provenant du code de votre fonction. Pour de plus amples informations, veuillez consulter Envoi automatique des journaux de la fonction Lambda à CloudWatch Logs.

Cette page explique comment générer une sortie de journal à partir du code de votre fonction Lambda et comment accéder aux journaux à l'aide de la AWS Command Line Interface console Lambda ou de la console. CloudWatch

Sections

- Création d'une fonction qui renvoie des journaux
- Affichage des journaux dans la console Lambda
- Afficher les journaux dans la CloudWatch console
- Afficher les journaux à l'aide de AWS Command Line Interface (AWS CLI)
- Suppression de journaux

Création d'une fonction qui renvoie des journaux

Pour générer des journaux à partir de votre code de fonction, vous pouvez utiliser des applets de commande sur Microsoft. PowerShell.Utility, ou tout module de journalisation qui écrit dans `stdout` ou`stderr`. L'exemple suivant utilise `Write-Host`.

Example function/Handler.ps1 – Journalisation

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host `## Environment variables
Write-Host AWS_LAMBDA_FUNCTION_VERSION=$Env:AWS_LAMBDA_FUNCTION_VERSION
Write-Host AWS_LAMBDA_LOG_GROUP_NAME=$Env:AWS_LAMBDA_LOG_GROUP_NAME
Write-Host AWS_LAMBDA_LOG_STREAM_NAME=$Env:AWS_LAMBDA_LOG_STREAM_NAME
Write-Host AWS_EXECUTION_ENV=$Env:AWS_EXECUTION_ENV
Write-Host AWS_LAMBDA_FUNCTION_NAME=$Env:AWS_LAMBDA_FUNCTION_NAME
Write-Host PATH=$Env:PATH
Write-Host `## Event
```

```
Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 3)
```

Example format des journaux

```
START RequestId: 56639408-xmpl-435f-9041-ac47ae25ceed Version: $LATEST
Importing module ./Modules/AWSPowerShell.NetCore/3.3.618.0/AWSPowerShell.NetCore.psd1
[Information] - ## Environment variables
[Information] - AWS_LAMBDA_FUNCTION_VERSION=$LATEST
[Information] - AWS_LAMBDA_LOG_GROUP_NAME=/aws/lambda/blank-powershell-
function-18CIXMPLHFAJJ
[Information] - AWS_LAMBDA_LOG_STREAM_NAME=2020/04/01/
[$LATEST]53c5xmpl52d64ed3a744724d9c201089
[Information] - AWS_EXECUTION_ENV=AWS_Lambda_dotnet6_powershell_1.0.0
[Information] - AWS_LAMBDA_FUNCTION_NAME=blank-powershell-function-18CIXMPLHFAJJ
[Information] - PATH=/var/lang/bin:/usr/local/bin:/usr/bin/:/bin:/opt/bin
[Information] - ## Event
[Information] -
{
    "Records": [
        {
            "messageId": "19dd0b57-b21e-4ac1-bd88-01bbb068cb78",
            "receiptHandle": "MessageReceiptHandle",
            "body": "Hello from SQS!",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1523232000000",
                "SenderId": "123456789012",
                "ApproximateFirstReceiveTimestamp": "1523232000001"
            },
            ...
END RequestId: 56639408-xmpl-435f-9041-ac47ae25ceed
REPORT RequestId: 56639408-xmpl-435f-9041-ac47ae25ceed Duration: 3906.38 ms Billed
 Duration: 9867 ms Memory Size: 512 MB Max Memory Used: 367 MB Init Duration: 5960.19
 ms
XRAY TraceId: 1-5e843da6-733cxmple7d0c3c020510040 SegmentId: 3913xmpl20999446 Sampled:
 true
```

L’environnement d’exécution .NET enregistre les lignes `START`, `END` et `REPORT` pour chaque invocation. La ligne de rapport fournit les détails suivants.

Champs de données de la ligne REPORT

- RequestId— L'identifiant de demande unique pour l'invocation.

- Duration – Temps que la méthode de gestion du gestionnaire de votre fonction a consacré au traitement de l'événement.
- Billed Duration : temps facturé pour l'invocation.
- Memory Size – Quantité de mémoire allouée à la fonction.
- Max Memory Used – Quantité de mémoire utilisée par la fonction. Lorsque les appels partagent un environnement d'exécution, Lambda indique la mémoire maximale utilisée pour toutes les invocations. Ce comportement peut entraîner une valeur signalée plus élevée que prévu.
- Init Duration : pour la première requête servie, temps qu'il a pris à l'exécution charger la fonction et exécuter le code en dehors de la méthode du gestionnaire.
- XRAY TraceId — Pour les demandes suivies, l'ID de AWS X-Ray trace.
- SegmentId— Pour les demandes tracées, l'identifiant du segment X-Ray.
- Sampled – Pour les demandes suivies, résultat de l'échantillonnage.

Affichage des journaux dans la console Lambda

Vous pouvez utiliser la console Lambda pour afficher la sortie du journal après avoir invoqué une fonction Lambda.

Si votre code peut être testé à partir de l'éditeur Code intégré, vous trouverez les journaux dans les résultats d'exécution. Lorsque vous utilisez la fonctionnalité de test de console pour invoquer une fonction, vous trouverez Sortie du journal dans la section Détails.

Afficher les journaux dans la CloudWatch console

Vous pouvez utiliser la CloudWatch console Amazon pour consulter les journaux de toutes les invocations de fonctions Lambda.

Pour afficher les journaux sur la CloudWatch console

1. Ouvrez la page Groupes de journaux sur la CloudWatch console.
2. Choisissez le groupe de journaux pour votre fonction (***your-function-name***/aws/lambda/).
3. Choisissez un flux de journaux.

Chaque flux de journal correspond à une instance de votre fonction. Un flux de journal apparaît lorsque vous mettez à jour votre fonction Lambda et lorsque des instances supplémentaires sont créées pour gérer les appels simultanés. Pour trouver les journaux d'un appel spécifique, nous vous

recommandons d'instrumenter votre fonction avec. AWS X-Ray X-Ray enregistre des détails sur la demande et le flux de journaux dans le suivi.

Afficher les journaux à l'aide de AWS Command Line Interface (AWS CLI)

AWS CLI Il s'agit d'un outil open source qui vous permet d'interagir avec les AWS services à l'aide de commandes dans votre interface de ligne de commande. Pour effectuer les étapes de cette section, vous devez disposer de la version 2 de l'AWS CLI.

Vous pouvez utiliser AWS CLI pour récupérer les journaux d'une invocation à l'aide de l'option de commande `--log-type`. La réponse inclut un champ `LogResult` qui contient jusqu'à 4 Ko de journaux codés en base64 provenant de l'invocation.

Example récupérer un ID de journal

L'exemple suivant montre comment récupérer un ID de journal à partir du champ `LogResult` d'une fonction nommée `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Vous devriez voir la sortie suivante:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example décoder les journaux

Dans la même invite de commandes, utilisez l'utilitaire `base64` pour décoder les journaux. L'exemple suivant montre comment récupérer les journaux encodés en base64 pour `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-`

`in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

Vous devriez voir la sortie suivante :

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

L'utilitaire `base64` est disponible sous Linux, macOS et Ubuntu sous Windows. Les utilisateurs de macOS auront peut-être besoin d'utiliser `base64 -D`.

Example Script get-logs.sh

Dans la même invite de commandes, utilisez le script suivant pour télécharger les cinq derniers événements de journalisation. Le script utilise `sed` pour supprimer les guillemets du fichier de sortie et attend 15 secondes pour permettre la mise à disposition des journaux. La sortie comprend la réponse de Lambda, ainsi que la sortie de la commande `get-log-events`.

Copiez le contenu de l'exemple de code suivant et enregistrez-le dans votre répertoire de projet Lambda sous `get-logs.sh`.

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS et Linux (uniquement)

Dans la même invite de commandes, les utilisateurs macOS et Linux peuvent avoir besoin d'exécuter la commande suivante pour s'assurer que le script est exécutable.

```
chmod -R 755 get-logs.sh
```

Example récupérer les cinq derniers événements de journal

Dans la même invite de commande, exécutez le script suivant pour obtenir les cinq derniers événements de journalisation.

```
./get-logs.sh
```

Vous devriez voir la sortie suivante:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
```

```
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


Suppression de journaux

Les groupes de journaux ne sont pas supprimés automatiquement quand vous supprimez une fonction. Pour éviter de stocker des journaux indéfiniment, supprimez le groupe de journaux ou configurez une période de conservation à l'issue de laquelle les journaux sont supprimés automatiquement.

Création de fonctions Lambda avec Rust

Étant donné que Rust se compile en code natif, vous n'avez pas besoin d'une d'exécution dédiée pour exécuter du code Rust sur Lambda. Utilisez plutôt le client d'exécution Rust pour créer votre projet localement, puis déployez-le sur Lambda à l'aide d'un environnement d'exécution uniquement pour le système d'exploitation. Lorsque vous utilisez un environnement d'exécution uniquement pour le système d'exploitation, Lambda tient automatiquement le système d'exploitation à jour avec les derniers correctifs.

Outils et bibliothèques pour Rust

- AWS SDK pour Rust: Le AWS SDK pour Rust permet à Rust APIs d'interagir avec les services d'infrastructure Amazon Web Services.
- Client d'exécution Rust pour Lambda : Le client d'exécution Rust facilite l'exécution de fonctions Lambda écrites en Rust.
- Cargo Lambda : il s'agit d'une extension open source tierce de l'outil de ligne de commande Cargo qui simplifie la création et le déploiement des fonctions Rust Lambda.
- Lambda HTTP : cette bibliothèque fournit un wrapper pour travailler avec des événements HTTP.
- Extension Lambda : cette bibliothèque fournit un support pour écrire des extensions Lambda avec Rust.
- AWS Lambda Événements : cette bibliothèque fournit des définitions de type pour les intégrations de sources d'événements courantes.

Exemples d'applications Lambda pour Rust

- Fonction Lambda de base : une fonction Rust qui montre comment traiter des événements de base.
- Fonction Lambda avec gestion des erreurs : une fonction Rust qui montre comment gérer des erreurs Rust personnalisées dans Lambda.
- Fonction Lambda avec ressources partagées : un projet Rust qui initialise les ressources partagées avant de créer la fonction Lambda.
- Événements HTTP Lambda : une fonction Rust qui gère les événements HTTP.
- Événements HTTP Lambda avec en-têtes CORS : une fonction Rust qui utilise Tower pour injecter des en-têtes CORS.

- API REST Lambda : une API REST qui utilise Axum et Diesel pour la connexion à une base de données PostgreSQL.
- Démo Rust sans serveur : un projet Rust qui montre l'utilisation des bibliothèques Rust de Lambda, de la journalisation, des variables d'environnement et du AWS SDK.
- Extension Lambda de base : une extension Rust qui montre comment traiter les événements d’extension de base.
- Extension Lambda Logs Amazon Data Firehose : une extension Rust qui montre comment envoyer les journaux Lambda à Firehose.

Rubriques

- Définir des gestionnaires de fonctions Lambda en Rust
- Utilisation de l’objet de contexte Lambda pour récupérer les informations de la fonction Rust
- Traitement des événements HTTP avec Rust
- Déploiement de fonctions Lambda Rust avec des archives de fichiers .zip
- Utilisation de couches pour les fonctions Lambda Rust
- Journalisation et surveillance des fonctions Lambda Rust

Définir des gestionnaires de fonctions Lambda en Rust

Le gestionnaire de fonction Lambda est la méthode dans votre code de fonction qui traite les événements. Lorsque votre fonction est invoquée, Lambda exécute la méthode du gestionnaire. Votre fonction s'exécute jusqu'à ce que le gestionnaire renvoie une réponse, se ferme ou expire.

Cette page explique comment utiliser les gestionnaires de fonctions Lambda dans Rust, notamment l'initialisation du projet, les conventions de dénomination et les pratiques exemplaires. Cette page inclut également un exemple de fonction Lambda Rust qui collecte des informations relatives à une commande, produit un reçu sous forme de fichier texte et place ce fichier dans un compartiment Amazon Simple Storage Service (S3). Pour plus d'informations sur le déploiement de votre fonction après l'avoir écrite, consultez the section called "Déploiement d'archives de fichiers .zip".

Rubriques

- Configuration de votre projet de gestionnaire Rust
- Exemple de fonction Lambda Rust
- Définitions de classe valides pour les gestionnaires Rust
- Convention de nommage du gestionnaire
- Définition et accès à l'objet d'événement d'entrée
- Accès et utilisation de l'objet de contexte Lambda
- En utilisant le AWS SDK pour Rust dans votre gestionnaire
- Accès aux variables d'environnement
- Utilisation de l'état partagé
- Pratiques exemplaires en matière de code pour les fonctions Lambda Rust

Configuration de votre projet de gestionnaire Rust

Lorsque vous utilisez des fonctions Lambda en Rust, le processus consiste à écrire votre code, à le compiler et à déployer les artefacts compilés sur Lambda. Le moyen le plus simple de configurer un projet de gestionnaire Lambda dans Rust est d'utiliser l'environnement d'exécution AWS Lambda pour Rust. Malgré son nom, le AWS Lambda Runtime for Rust n'est pas un environnement d'exécution géré au même sens que dans Lambda pour Python, Java ou Node.js. Au lieu de cela, le AWS Lambda Runtime for Rust est un crate (`lambda_runtime`) qui permet d'écrire des fonctions Lambda dans Rust et de s'interfacer avec l'environnement d'exécution AWS Lambda de celui-ci.

Utilisez la commande suivante pour installer Cargo Lambda, une extension open source tierce de l'outil de ligne de commande Cargo qui simplifie la création et le déploiement des fonctions Rust Lambda :

```
cargo install cargo-lambda
```

Après avoir installé `cargo-lambda`, utilisez la commande suivante pour initialiser un nouveau projet de gestionnaire de fonction Lambda :

```
cargo lambda new example-rust
```

Lorsque vous exécutez cette commande, l'interface de ligne de commande (CLI) vous pose quelques questions concernant votre fonction Lambda :

- Fonction HTTP : si vous avez l'intention d'invoquer votre fonction via API Gateway ou une URL de fonction, répondez Oui. Dans le cas contraire, répondez Non. Dans l'exemple de code de cette page, nous invoquons notre fonction avec un événement JSON personnalisé. Nous répondons donc Non.
- Type d'événement : si vous avez l'intention d'utiliser une forme d'événement prédéfinie pour invoquer votre fonction, sélectionnez le type d'événement attendu approprié. Sinon, laissez cette option vide. Dans l'exemple de code de cette page, nous invoquons notre fonction avec un événement JSON personnalisé. Nous laissons donc cette option vide.

Une fois la commande exécutée avec succès, entrez dans le répertoire principal de votre projet :

```
cd example-rust
```

Cette commande génère un fichier `generic_handler.rs` et un fichier `main.rs` dans le répertoire `src`. `generic_handler.rs` peut être utilisé pour personnaliser un gestionnaire d'événements générique. Le fichier `main.rs` contient la logique de votre application principale. Le fichier `Cargo.toml` contient des métadonnées relatives à votre package et répertorie ses dépendances externes.

Exemple de fonction Lambda Rust

L'exemple de code de fonction Lambda Rust suivant prend en compte les informations relatives à une commande, produit un reçu sous forme de fichier texte et place ce fichier dans un compartiment Amazon S3.

Example Fonction Lambda **`main.rs`**

```
use aws_sdk_s3::{Client, primitives::ByteStream};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
use serde::{Deserialize, Serialize};
use serde_json::Value;
use std::env;

#[derive(Deserialize, Serialize)]
struct Order {
    order_id: String,
    amount: f64,
    item: String,
}

async fn function_handler(event: LambdaEvent<Value>) -> Result<String, Error> {
    let payload = event.payload;

    // Deserialize the incoming event into Order struct
    let order: Order = serde_json::from_value(payload)?;

    let bucket_name = env::var("RECEIPT_BUCKET")
        .map_err(|_| "RECEIPT_BUCKET environment variable is not set")?;

    let receipt_content = format!(
        "OrderID: {}\nAmount: ${:.2}\nItem: {}",
        order.order_id, order.amount, order.item
    );
    let key = format!("receipts/{}.txt", order.order_id);

    let config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
    let s3_client = Client::new(&config);

    upload_receipt_to_s3(&s3_client, &bucket_name, &key, &receipt_content).await?;

    Ok("Success".to_string())
}

async fn upload_receipt_to_s3(
    client: &Client,
    bucket_name: &str,
    key: &str,
    content: &str,
```

```
) -> Result<(), Error> {
    client
        .put_object()
        .bucket(bucket_name)
        .key(key)
        .body(ByteStream::from(content.as_bytes().to_vec()))  // Fixed conversion
        .content_type("text/plain")
        .send()
        .await?;

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(function_handler)).await
}
```

Ce fichier `main.rs` comprend les sections suivantes :

- Instructions `use` : utilisez-les pour importer des conteneurs (crates) et méthodes Rust que votre fonction Lambda requiert.
- `#[derive(Deserialize, Serialize)]` : définissez la forme de l'événement d'entrée attendu dans cette structure Rust.
- `async fn function_handler(event: LambdaEvent<Value>) -> Result<String, Error>` : il s'agit de la méthode de gestion principale, qui contient la logique principale de votre application.
- `async fn upload_receipt_to_s3 (...)` : il s'agit d'une méthode auxiliaire référencée par la méthode `function_handler` principale.
- `#[tokio::main]` : il s'agit d'une macro qui marque le point d'entrée d'un programme Rust. Elle met également en place un environnement d'exécution Tokio, qui permet à votre méthode `main()` d'utiliser `async`/`await` et de s'exécuter de manière asynchrone.
- `async fn main() -> Result<(), Error>` : la fonction `main()` est le point d'entrée de votre code. À l'intérieur, nous spécifions `function_handler` comme méthode de gestionnaire principal.

Exemple de fichier Cargo.toml

Le fichier `Cargo.toml` suivant accompagne cette fonction.

```
[package]
name = "example-rust"
version = "0.1.0"
edition = "2024"

[dependencies]
aws-config = "1.5.18"
aws-sdk-s3 = "1.78.0"
lambda_runtime = "0.13.0"
serde = { version = "1", features = ["derive"] }
serde_json = "1"
tokio = { version = "1", features = ["full"] }
```

Pour que cette fonction fonctionne correctement, son rôle d'exécution doit autoriser l'action `s3:PutObject`. Assurez-vous également de définir la variable d'environnement `RECEIPT_BUCKET`. Après une invocation réussie, le compartiment Amazon S3 doit contenir un fichier de reçu.

Définitions de classe valides pour les gestionnaires Rust

Dans la plupart des cas, les signatures du gestionnaire Lambda que vous définissez dans Rust auront le format suivant :

```
async fn function_handler(event: LambdaEvent<T>) -> Result<U, Error>
```

Pour ce gestionnaire :

- Le nom de ce gestionnaire est `function_handler`.
- L'entrée singulière du gestionnaire est un événement, et est de type `LambdaEvent<T>`.
  - `LambdaEvent` est un wrapper qui vient du conteneur (crate) `lambda_runtime`. L'utilisation de ce wrapper vous donne accès à l'objet de contexte, qui inclut des métadonnées spécifiques à Lambda, comme l'ID de demande de l'invocation.
  - T est le type d'événement désérialisé. Par exemple, cela peut être `serde_json::Value`, qui permet au gestionnaire d'accepter n'importe quelle entrée JSON générique. Il peut également s'agir d'un type comme `ApiGatewayProxyRequest` si votre fonction attend un type d'entrée spécifique et prédéfini.
- Le type de retour du gestionnaire est `Result<U, Error>`.
  - U est le type de sortie désérialisé. U doit implémenter la caractéristique `serde::Serialize` afin que Lambda puisse convertir la valeur de retour en JSON. Par exemple, U peut être un

type simple comme `String`, `serde_json::Value` ou une structure personnalisée tant qu'il implémente `Serialize`. Lorsque votre code atteint une instruction Ok(U), cela indique une exécution réussie et votre fonction renvoie une valeur de type U.

- Lorsque votre code rencontre une erreur (c'est-à-dire`Err(Error)`), votre fonction enregistre l'erreur dans Amazon CloudWatch et renvoie une réponse d'erreur de type`Error`.

Dans notre exemple, la signature du gestionnaire ressemble à ce qui suit :

```
async fn function_handler(event: LambdaEvent<Value>) -> Result<String, Error>
```

Les autres signatures de gestionnaire valides peuvent présenter les caractéristiques suivantes :

- Omission du wrapper `LambdaEvent` : si vous omettez `LambdaEvent`, vous perdez l'accès à l'objet de contexte Lambda dans votre fonction. Voici un exemple de ce type de signature :

  ```
  async fn handler(event: serde_json::Value) -> Result<String, Error>
  ```

- Utilisation du type d'unité comme entrée : pour Rust, vous pouvez utiliser le type d'unité pour représenter une entrée vide. Cela est couramment utilisé pour les fonctions avec des invocations périodiques et planifiées. Voici un exemple de ce type de signature :

  ```
  async fn handler(_: ()) -> Result<Value, Error>
  ```

Convention de nommage du gestionnaire

Les gestionnaires Lambda dans Rust n'ont pas de restrictions de dénomination strictes. Bien que vous puissiez utiliser n'importe quel nom pour votre gestionnaire, les noms de fonctions en Rust suivent généralement le format de casse `snake_case`.

Pour les applications plus petites, comme dans cet exemple, vous pouvez utiliser un seul fichier `main.rs` pour contenir l'ensemble de votre code. Pour les projets plus importants, `main.rs` doit contenir le point d'entrée de votre fonction, mais vous pouvez avoir des fichiers supplémentaires pour séparer votre code en modules logiques. Par exemple, vous pouvez avoir la structure de fichiers suivante :

```
/example-rust
### src/
```

```
#   ### main.rs        # Entry point
#   ### handler.rs     # Contains main handler
#   ### services.rs    # [Optional] Back-end service calls
#   ### models.rs      # [Optional] Data models
### Cargo.toml
```

Définition et accès à l'objet d'événement d'entrée

JSON est le format d'entrée le plus courant et standard pour les fonctions Lambda. Dans cet exemple, la fonction exige une entrée similaire à l'exemple suivant :

```
{
    "order_id": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

En Rust, vous pouvez définir la forme de l'événement d'entrée attendu dans une structure. Dans cet exemple, nous définissons la structure suivante pour représenter un `Order` :

```
#[derive(Deserialize, Serialize)]
struct Order {
    order_id: String,
    amount: f64,
    item: String,
}
```

Cette structure correspond à la forme d'entrée attendue. Dans cet exemple, la macro `#[derive(Deserialize, Serialize)]` génère automatiquement du code pour la sérialisation et la désérialisation. Cela signifie que nous pouvons désérialiser le type JSON d'entrée générique dans notre structure à l'aide de la méthode `serde_json::from_value()`. Ceci est illustré dans les premières lignes du gestionnaire :

```
async fn function_handler(event: LambdaEvent<Value>) -> Result<String, Error> {
    let payload = event.payload;

    // Deserialize the incoming event into Order struct
    let order: Order = serde_json::from_value(payload)?;
    ...
}
```

Vous pouvez ensuite accéder aux champs de l'objet. Par exemple, `order.order_id` récupère la valeur de `order_id` à partir de l'entrée d'origine.

Types d'événements d'entrée prédéfinis

De nombreux types d'événements d'entrée prédéfinis sont disponibles dans le conteneur (crate) `aws_lambda_events`. Par exemple, si vous avez l'intention d'invoquer votre fonction avec API Gateway, y compris l'importation suivante :

```
use aws_lambda_events::event::apigw::ApiGatewayProxyRequest;
```

assurez-vous que votre gestionnaire principal utilise la signature suivante :

```
async fn handler(event: LambdaEvent<ApiGatewayProxyRequest>) -> Result<String, Error> {
    let body = event.payload.body.unwrap_or_default();
    ...
}
```

Reportez-vous au conteneur aws_lambda_events pour plus d'informations sur les autres types d'événements d'entrée prédéfinis.

Accès et utilisation de l'objet de contexte Lambda

L'objet de contexte Lambda contient des informations sur l'invocation, la fonction et l'environnement d'exécution. En Rust, le wrapper `LambdaEvent` inclut l'objet de contexte. Par exemple, vous pouvez utiliser l'objet de contexte pour récupérer l'ID de demande de l'invocation en cours à l'aide du code suivant :

```
async fn function_handler(event: LambdaEvent<Value>) -> Result<String, Error> {
    let request_id = event.context.request_id;
    ...
}
```

Pour plus d'informations sur la copie d'objets, consultez the section called "Contexte".

En utilisant le AWS SDK pour Rust dans votre gestionnaire

Vous utiliserez souvent les fonctions Lambda pour interagir avec d'autres AWS ressources ou pour les mettre à jour. Le moyen le plus simple d'interagir avec ces ressources est d'utiliser AWS SDK pour Rust.

Pour ajouter des dépendances du SDK à votre fonction, ajoutez-les dans votre fichier `Cargo.toml`. Nous vous recommandons de n'ajouter que les bibliothèques dont vous avez besoin pour votre fonction. Dans l'exemple de code précédent, nous avons utilisé `aws_sdk_s3::Client`. Dans le fichier `Cargo.toml`, vous pouvez ajouter cette dépendance en ajoutant la ligne suivante sous la section `[dependencies]` :

```
aws-sdk-s3 = "1.78.0"
```

> **Note**
>
> La version n'est peut-être pas la plus récente. Choisissez la version adéquate pour votre application.

Importez ensuite les dépendances directement dans votre code :

```
use aws_sdk_s3::{Client, primitives::ByteStream};
```

L'exemple de code initialise ensuite un client Amazon S3 comme suit :

```
let config = aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let s3_client = Client::new(&config);
```

Après avoir initialisé votre client SDK, vous pouvez l'utiliser pour interagir avec d'autres AWS services. L'exemple de code appelle l'API `PutObject` Amazon S3 dans la fonction d'assistance `upload_receipt_to_s3`.

Accès aux variables d'environnement

Dans le code de votre gestionnaire, vous pouvez référencer n'importe quelle variable d'environnement à l'aide de la méthode `env::var`. Dans cet exemple, nous référençons la variable d'environnement `RECEIPT_BUCKET` définie à l'aide de la ligne de code suivante :

```
let bucket_name = env::var("RECEIPT_BUCKET")
    .map_err(|_| "RECEIPT_BUCKET environment variable is not set")?;
```

Utilisation de l'état partagé

Vous pouvez déclarer des variables partagées indépendantes du code du gestionnaire de votre fonction Lambda. Ces variables peuvent vous aider à charger des informations d'état pendant le Phase d'initialisation, avant que votre fonction ne reçoive des événements. Par exemple, vous pouvez modifier le code de cette page pour utiliser l'état partagé lors de l'initialisation du client Amazon S3 en mettant à jour la fonction `main` et la signature du gestionnaire :

```
async fn function_handler(client: &Client, event: LambdaEvent<Value>) -> Result<String,
 Error> {
    ...
    upload_receipt_to_s3(client, &bucket_name, &key, &receipt_content).await?;
    ...
}

...

#[tokio::main]
async fn main() -> Result<(), Error> {
    let shared_config = aws_config::from_env().load().await;
    let client = Client::new(&shared_config);
    let shared_client = &client;
    lambda_runtime::run(service_fn(move |event: LambdaEvent<Request>| async move {
        handler(&shared_client, event).await
    }))
    .await
```

Pratiques exemplaires en matière de code pour les fonctions Lambda Rust

Respectez les directives de la liste suivante pour utiliser les pratiques exemplaires de codage lors de la création de vos fonctions Lambda :

- Séparez le gestionnaire Lambda de votre logique principale. Cela vous permet de créer une fonction testable plus unitaire.
- Réduisez la complexité de vos dépendances. Privilégiez les infrastructures plus simples qui se chargent rapidement au démarrage de l'environnement d'exécution.
- Réduisez la taille de votre package de déploiement selon ses besoins d'exécution. Cela contribue à réduire le temps nécessaire au téléchargement et à la décompression de votre package de déploiement avant l'invocation.

Tirez parti de la réutilisation de l'environnement d'exécution pour améliorer les performances de votre fonction. Initialisez les clients SDK et les connexions à la base de données en dehors du gestionnaire de fonctions et mettez en cache les actifs statiques localement dans le répertoire `/tmp`. Les invocations ultérieures traitées par la même instance de votre fonction peuvent réutiliser ces ressources. Cela permet d'économiser des coûts, tout en réduisant le temps d'exécution de la fonction.

Pour éviter des éventuelles fuites de données entre les invocations, n'utilisez pas l'environnement d'exécution pour stocker des données utilisateur, des événements ou d'autres informations ayant un impact sur la sécurité. Si votre fonction repose sur un état réversible qui ne peut pas être stocké en mémoire dans le gestionnaire, envisagez de créer une fonction distincte ou des versions distinctes d'une fonction pour chaque utilisateur.

Utilisez une directive keep-alive pour maintenir les connexions persistantes. Lambda purge les connexions inactives au fil du temps. Si vous tentez de réutiliser une connexion inactive lorsque vous invoquez une fonction, cela entraîne une erreur de connexion. Pour maintenir votre connexion persistante, utilisez la directive Keep-alive associée à votre environnement d'exécution. Pour obtenir un exemple, consultez Réutilisation des connexions avec Keep-Alive dans Node.js.

Utilisez des variables d'environnement pour transmettre des paramètres opérationnels à votre fonction. Par exemple, si vous écrivez dans un compartiment Amazon S3 au lieu de coder en dur le nom du compartiment dans lequel vous écrivez, configurez le nom du compartiment comme variable d'environnement.

Évitez d'utiliser des invocations récursives dans votre fonction Lambda, lorsque la fonction s'invoque elle-même ou démarre un processus susceptible de l'invoquer à nouveau. Cela peut entraîner un volume involontaire d'invocations de fonction et des coûts accrus. Si vous constatez un volume involontaire d'invocations, définissez immédiatement la simultanéité réservée à la fonction sur `0` afin de limiter toutes les invocations de la fonction, pendant que vous mettez à jour le code.

N'utilisez pas de code non documenté ni public APIs dans votre code de fonction Lambda. Pour les AWS Lambda environnements d'exécution gérés, Lambda applique régulièrement des mises à jour de sécurité et fonctionnelles aux applications internes de Lambda. APIs Ces mises à jour internes de l'API peuvent être rétroincompatibles, ce qui peut entraîner des conséquences imprévues, telles que des échecs d'invocation si votre fonction dépend de ces mises à jour non publiques. APIs Consultez la référence de l'API pour obtenir une liste des API accessibles au public APIs.

Écriture du code idempotent. L'écriture de code idempotent pour vos fonctions garantit ne gestion identique des événements dupliqués. Votre code doit valider correctement les événements

et gérer correctement les événements dupliqués. Pour de plus amples informations, veuillez consulterComment faire en sorte que ma fonction Lambda soit idempotente ?.

Utilisation de l’objet de contexte Lambda pour récupérer les informations de la fonction Rust

Lorsque Lambda exécute votre fonction, il ajoute un objet de contexte à LambdaEvent celui que le gestionnaire reçoit. Cet objet fournit les propriétés avec des informations sur l’appel, la fonction et l’environnement d’exécution.

Propriétés du contexte

- `request_id`: ID de AWS demande généré par le service Lambda.
- `deadline` : le délai d’exécution de l’invocation en cours en millisecondes.
- `invoked_function_arn` : l’Amazon Resource Name (ARN) de la fonction Lambda en cours d’invocation.
- `xray_trace_id`: ID de AWS X-Ray trace pour l'appel en cours.
- `client_content`: objet de contexte client envoyé par le SDK AWS mobile. Ce champ est vide sauf si la fonction est invoquée à l'aide d'un SDK AWS mobile.
- `identity` : l’identité Amazon Cognito qui a invoqué la fonction. Ce champ est vide sauf si la demande d'invocation adressée au APIs Lambda a été faite à l' AWS aide d'informations d'identification émises par les groupes d'identités Amazon Cognito.
- `env_config` : la configuration de la fonction Lambda à partir des variables d’environnement locales. Cette propriété comprend des informations telles que le nom de la fonction, l’allocation de mémoire, la version et les flux de journaux.

Accès aux informations du contexte d’appel

Les fonctions Lambda ont accès aux métadonnées sur leur environnement et la demande d’appel. L’objet `LambaEvent` que votre gestionnaire de fonction reçoit comprend les métadonnées `context` :

```
use lambda_runtime::{service_fn, LambdaEvent, Error};
use serde_json::{json, Value};

async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    let invoked_function_arn = event.context.invoked_function_arn;
    Ok(json!({ "message": format!("Hello, this is function
 {invoked_function_arn}!") }))
}
```

```
#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

Traitement des événements HTTP avec Rust

Amazon API Gateway APIs, les équilibreurs de charge d'application et la URLs fonction Lambda peuvent envoyer des événements HTTP à Lambda. Vous pouvez utiliser la caisse aws_lambda_events de crates.io pour traiter les événements provenant de ces sources.

Example – Gestion de la demande de proxy API Gateway

Notez ce qui suit :

- `use aws_lambda_events::apigw::{ApiGatewayProxyRequest, ApiGatewayProxyResponse}` : la caisse aws_lambda_events inclut de nombreux événements Lambda. Pour réduire le temps de compilation, utilisez les drapeaux de fonction pour activer les événements dont vous avez besoin. Exemple: `aws_lambda_events = { version = "0.8.3", default-features = false, features = ["apigw"] }`.
- `use http::HeaderMap` : cette importation nécessite d'ajouter la caisse http à vos dépendances.

```
use aws_lambda_events::apigw::{ApiGatewayProxyRequest, ApiGatewayProxyResponse};
use http::HeaderMap;
use lambda_runtime::{service_fn, Error, LambdaEvent};

async fn handler(
    _event: LambdaEvent<ApiGatewayProxyRequest>,
) -> Result<ApiGatewayProxyResponse, Error> {
    let mut headers = HeaderMap::new();
    headers.insert("content-type", "text/html".parse().unwrap());
    let resp = ApiGatewayProxyResponse {
        status_code: 200,
        multi_value_headers: headers.clone(),
        is_base64_encoded: false,
        body: Some("Hello AWS Lambda HTTP request".into()),
        headers,
    };
    Ok(resp)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

Le client d'exécution Rust pour Lambda fournit également une abstraction sur ces types d'événements qui vous permet de travailler avec les types HTTP natifs, quel que soit le service qui envoie les événements. Le code suivant est équivalent à l'exemple précédent et fonctionne immédiatement avec la fonction Lambda URLs, les équilibreurs de charge d'application et l'API Gateway.

> **Note**
>
> À la base, la caisse lambda_http utilise la caisse lambda_runtime. Vous n'avez pas besoin d'importer `lambda_runtime` séparément.

Example – Gestion des requêtes HTTP

```
use lambda_http::{service_fn, Error, IntoResponse, Request, RequestExt, Response};

async fn handler(event: Request) -> Result<impl IntoResponse, Error> {
    let resp = Response::builder()
        .status(200)
        .header("content-type", "text/html")
        .body("Hello AWS Lambda HTTP request")
        .map_err(Box::new)?;
    Ok(resp)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_http::run(service_fn(handler)).await
}
```

Pour un autre exemple d'utilisation`lambda_http`, consultez l'exemple de code http-axum sur le AWS référentiel Labs. GitHub

Exemples d'événements HTTP Lambda pour Rust

- Événements HTTP Lambda : une fonction Rust qui gère les événements HTTP.
- Événements HTTP Lambda avec en-têtes CORS : une fonction Rust qui utilise Tower pour injecter des en-têtes CORS.
- Événements HTTP Lambda avec ressources partagées : une fonction Rust qui utilise des ressources partagées initialisées avant la création du gestionnaire de fonction.

Déploiement de fonctions Lambda Rust avec des archives de fichiers .zip

Cette page décrit comment compiler votre fonction Rust, puis déployer le binaire compilé à l' AWS Lambda aide de Cargo Lambda. Il montre également comment déployer le binaire compilé à l'aide de la CLI AWS Command Line Interface et de la AWS Serverless Application Model CLI.

Sections

- Conditions préalables
- Création de fonctions Rust sur macOS, Windows ou Linux
- Déploiement du binaire de la fonction Rust avec Cargo Lambda
- Invocation de votre fonction Rust avec Cargo Lambda

Conditions préalables

- Rust
- AWS CLI version 2

Création de fonctions Rust sur macOS, Windows ou Linux

Les étapes suivantes montrent comment créer le projet de votre première fonction Lambda avec Rust et le compiler avec Cargo Lambda, une extension open source tierce de l'outil de ligne de commande Cargo qui simplifie la création et le déploiement des fonctions Rust Lambda.

1. Installez Cargo Lambda, une extension open source tierce de l'outil de ligne de commande Cargo qui simplifie la création et le déploiement des fonctions Rust Lambda :

   ```
   cargo install cargo-lambda
   ```

   Pour les autres options d'installation, consultez la section Installation dans la documentation de Cargo Lambda.
2. Créez la structure du package. Cette commande crée un code de fonction de base dans `src/main.rs`. Vous pouvez utiliser ce code pour le tester ou le remplacer par le vôtre.

   ```
   cargo lambda new my-function
   ```

3. À l'intérieur du répertoire racine du package, exécutez la sous-commande build pour compiler le code de votre fonction.

```
cargo lambda build --release
```

(Facultatif) Si vous souhaitez utiliser AWS Graviton2 sur Lambda, ajoutez l'`--arm64`indicateur pour compiler votre code pour ARM. CPUs

```
cargo lambda build --release --arm64
```

4. Avant de déployer votre fonction Rust, configurez les AWS informations d'identification sur votre machine.

```
aws configure
```

Déploiement du binaire de la fonction Rust avec Cargo Lambda

Utilisez la sous-commande deploy pour déployer le binaire compilé vers Lambda. Cette commande crée un rôle d'exécution, puis la fonction Lambda. Pour spécifier un rôle d'exécution existant, utilisez l'indicateur --iam-role.

```
cargo lambda deploy my-function
```

Déployer le binaire de votre fonction Rust avec le AWS CLI

Vous pouvez également déployer votre binaire avec le AWS CLI.

1. Utilisez la sous-commande build pour créer le package de déploiement .zip.

```
cargo lambda build --release --output-format zip
```

2. Pour déployer le package .zip vers Lambda, exécutez la commande create-function.

   - Pour `--runtime`, spécifiez `provided.al2023`. Il s'agit d'un environnement d'exécution réservé au système d'exploitation. Les environnements d'exécution réservés au système d'exploitation sont utilisés pour déployer des fichiers binaires compilés et des environnements d'exécution personnalisés sur Lambda.
   - Pour `--role`, spécifiez l'ARN du rôle d'exécution.

```
aws lambda create-function \
     --function-name my-function \
     --runtime provided.al2023 \
     --role arn:aws:iam::111122223333:role/lambda-role \
     --handler rust.handler \
     --zip-file fileb://target/lambda/my-function/bootstrap.zip
```

Déploiement de votre fonction binaire Rust avec la AWS SAM CLI

Vous pouvez également déployer votre binaire à l'aide de la AWS SAM CLI.

1. Créez un AWS SAM modèle avec la définition de la ressource et de la propriété. Pour `Runtime`, spécifiez `provided.al2023`. Il s'agit d'un environnement d'exécution réservé au système d'exploitation. Les environnements d'exécution réservés au système d'exploitation sont utilisés pour déployer des fichiers binaires compilés et des environnements d'exécution personnalisés sur Lambda.

   Pour plus d'informations sur le déploiement de fonctions Lambda à l'aide de fonctions Lambda AWS SAM, consultez AWS::Serverless::Functionle manuel du AWS Serverless Application Model développeur.

   Example Définition des ressources et des propriétés SAM pour un binaire Rust

   ```
   AWSTemplateFormatVersion: '2010-09-09'
   Transform: AWS::Serverless-2016-10-31
   Description: SAM template for Rust binaries
   Resources:
     RustFunction:
       Type: AWS::Serverless::Function
       Properties:
         CodeUri: target/lambda/my-function/
         Handler: rust.handler
         Runtime: provided.al2023
   Outputs:
     RustFunction:
       Description: "Lambda Function ARN"
       Value: !GetAtt RustFunction.Arn
   ```

2. Utilisez la sous-commande build pour compiler la fonction.

```
cargo lambda build --release
```

3. Utilisez la commande sam deploy pour déployer la fonction vers Lambda.

```
sam deploy --guided
```

Pour plus d'informations sur la création de fonctions Rust avec la AWS SAM CLI, consultez la section Création de fonctions Rust Lambda avec Cargo Lambda dans le manuel du développeur.AWS Serverless Application Model

Invocation de votre fonction Rust avec Cargo Lambda

Utilisez la sous-commande invoke pour tester votre fonction avec une charge utile.

```
cargo lambda invoke --remote --data-ascii '{"command": "Hello world"}' my-function
```

Invoquer votre fonction Rust avec le AWS CLI

Vous pouvez également utiliser le AWS CLI pour appeler la fonction.

```
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"command": "Hello world"}' /tmp/out.txt
```

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales prises en charge par l'AWS CLI dans le Guide de l'utilisateur AWS Command Line Interface version 2.

Utilisation de couches pour les fonctions Lambda Rust

Nous déconseillons d'utiliser des couches pour gérer les dépendances des fonctions Lambda écrites en Rust. Cela est dû au fait que les fonctions Lambda en Rust sont compilées en un seul exécutable, que vous fournissez à Lambda lorsque vous déployez votre fonction. Cet exécutable contient votre code de fonction compilé, ainsi que toutes ses dépendances. L'utilisation de couches complique non seulement ce processus, mais entraîne également une augmentation des temps de démarrage à froid, car vos fonctions doivent charger manuellement des assemblages supplémentaires en mémoire pendant la phase d'initialisation.

Pour utiliser des dépendances externes avec vos gestionnaires Rust, incluez-les directement dans votre package de déploiement. Ce faisant, vous simplifiez le processus de déploiement et profitez des optimisations du compilateur Rust intégré. Pour un exemple d'importation et d'utilisation d'une dépendance telle que le kit SDK AWS pour Rust dans votre fonction, consultez the section called "Handler (Gestionnaire)".

Journalisation et surveillance des fonctions Lambda Rust

AWS Lambda surveille automatiquement les fonctions Lambda en votre nom et envoie les journaux à Amazon. CloudWatch Votre fonction Lambda est fournie avec un groupe de CloudWatch journaux Logs et un flux de journaux pour chaque instance de votre fonction. L'environnement d'exécution Lambda envoie des informations sur chaque invocation au flux de journaux et transmet les journaux et autres sorties provenant du code de votre fonction. Pour de plus amples informations, veuillez consulter Envoi automatique des journaux de la fonction Lambda à CloudWatch Logs. Pour plus d'informations sur la configuration des formats de journal, consultezConfiguration des formats de journal JSON et en texte brut. Cette page décrit comment produire des journaux à partir du code de votre fonction Lambda.

Création d'une fonction qui écrit des journaux

Pour produire des journaux à partir du code de votre fonction, vous pouvez utiliser n'importe quelle fonction de journalisation qui écrit dans `stdout` ou `stderr`, comme la macro `println!`. L'exemple suivant utilise `println!` pour imprimer un message lorsque le gestionnaire de fonction démarre et avant qu'il ne se termine.

```
use lambda_runtime::{service_fn, LambdaEvent, Error};
use serde_json::{json, Value};
async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    println!("Rust function invoked");
    let payload = event.payload;
    let first_name = payload["firstName"].as_str().unwrap_or("world");
    println!("Rust function responds to {}", &first_name);
    Ok(json!({ "message": format!("Hello, {first_name}!") }))
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

Implémentation de la journalisation avancée avec la caisse Tracing

Tracing est un cadre permettant d'instrumenter les programmes Rust afin de collecter des informations de diagnostic structurées et basées sur des événements. Ce cadre fournit des utilitaires pour personnaliser les niveaux et les formats de sortie des journaux, comme la création de messages

de journal JSON structurés. Pour utiliser ce cadre, vous devez initialiser un `subscriber` avant de mettre en œuvre le gestionnaire de fonction. Ensuite, vous pouvez utiliser des macros de traçage comme `debug`, `info` et `error` pour spécifier le niveau de journal que vous voulez pour chaque scénario.

Example – Utilisation de la caisse Tracing

Notez ce qui suit :

- `tracing_subscriber::fmt().json()` : lorsque cette option est incluse, les journaux sont formatés en JSON. Pour utiliser cette option, vous devez inclure la fonction `json` dans la dépendance `tracing-subscriber` (par exemple, `tracing-subscriber = { version = "0.3.11", features = ["json"] }`).
- `#[tracing::instrument(skip(event), fields(req_id = %event.context.request_id))]` : cette annotation génère une étendue à chaque fois que le gestionnaire est invoqué. L'étendue ajoute l'identifiant de la demande à chaque ligne de journal.
- `{ %first_name }` : cette construction ajoute le champ `first_name` à la ligne de journal où il est utilisé. La valeur de ce champ correspond à la variable du même nom.

```
use lambda_runtime::{service_fn, Error, LambdaEvent};
use serde_json::{json, Value};
#[tracing::instrument(skip(event), fields(req_id = %event.context.request_id))]
async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    tracing::info!("Rust function invoked");
    let payload = event.payload;
    let first_name = payload["firstName"].as_str().unwrap_or("world");
    tracing::info!({ %first_name }, "Rust function responds to event");
    Ok(json!({ "message": format!("Hello, {first_name}!") }))
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt().json()
        .with_max_level(tracing::Level::INFO)
        // this needs to be set to remove duplicated information in the log.
        .with_current_span(false)
        // this needs to be set to false, otherwise ANSI color codes will
        // show up in a confusing manner in CloudWatch logs.
        .with_ansi(false)
        // disabling time is handy because CloudWatch will add the ingestion time.
```

```
        .without_time()
        // remove the name of the function from every log entry
        .with_target(false)
        .init();
    lambda_runtime::run(service_fn(handler)).await
}
```

Lorsque cette fonction Rust est invoquée, elle imprime deux lignes de journal similaires à ce qui suit :

```
{"level":"INFO","fields":{"message":"Rust function invoked"},"spans":
[{"req_id":"45daaaa7-1a72-470c-9a62-e79860044bb5","name":"handler"}]}
{"level":"INFO","fields":{"message":"Rust function responds to
 event","first_name":"David"},"spans":[{"req_id":"45daaaa7-1a72-470c-9a62-
e79860044bb5","name":"handler"}]}
```

Bonnes pratiques d'utilisation des AWS Lambda fonctions

Les bonnes pratiques suivantes sont recommandées pour l'utilisation d' AWS Lambda :

Rubriques

- Code de fonction
- Configuration de la fonction
- Capacité de mise à l'échelle de la fonction
- Métriques et alarmes
- Utilisation des flux
- Bonnes pratiques de sécurité

Code de fonction

Tirez parti de la réutilisation de l'environnement d'exécution pour améliorer les performances de votre fonction. Initialisez les clients SDK et les connexions à la base de données en dehors du gestionnaire de fonctions et mettez en cache les actifs statiques localement dans le répertoire `/tmp`. Les invocations ultérieures traitées par la même instance de votre fonction peuvent réutiliser ces ressources. Cela permet d'économiser des coûts, tout en réduisant le temps d'exécution de la fonction.

Pour éviter des éventuelles fuites de données entre les invocations, n'utilisez pas l'environnement d'exécution pour stocker des données utilisateur, des événements ou d'autres informations ayant un impact sur la sécurité. Si votre fonction repose sur un état réversible qui ne peut pas être stocké en mémoire dans le gestionnaire, envisagez de créer une fonction distincte ou des versions distinctes d'une fonction pour chaque utilisateur.

Utilisez une directive keep-alive pour maintenir les connexions persistantes. Lambda purge les connexions inactives au fil du temps. Si vous tentez de réutiliser une connexion inactive lorsque vous invoquez une fonction, cela entraîne une erreur de connexion. Pour maintenir votre connexion persistante, utilisez la directive Keep-alive associée à votre environnement d'exécution. Pour obtenir un exemple, consultez Réutilisation des connexions avec Keep-Alive dans Node.js.

Utilisez des variables d'environnement pour transmettre des paramètres opérationnels à votre fonction. Par exemple, si vous écrivez dans un compartiment Amazon S3 au lieu de coder en dur le

nom du compartiment dans lequel vous écrivez, configurez le nom du compartiment comme variable d'environnement.

Évitez d'utiliser des invocations récursives dans votre fonction Lambda, lorsque la fonction s'invoque elle-même ou démarre un processus susceptible de l'invoquer à nouveau. Cela peut entraîner un volume involontaire d'invocations de fonction et des coûts accrus. Si vous constatez un volume involontaire d'invocations, définissez immédiatement la simultanéité réservée à la fonction sur `0` afin de limiter toutes les invocations de la fonction, pendant que vous mettez à jour le code.

N'utilisez pas de code non documenté ni public APIs dans votre code de fonction Lambda. Pour les AWS Lambda environnements d'exécution gérés, Lambda applique régulièrement des mises à jour de sécurité et fonctionnelles aux applications internes de Lambda. APIs Ces mises à jour internes de l'API peuvent être rétroincompatibles, ce qui peut entraîner des conséquences imprévues, telles que des échecs d'invocation si votre fonction dépend de ces mises à jour non publiques. APIs Consultez la référence de l'API pour obtenir une liste des API accessibles au public APIs.

Écriture du code idempotent. L'écriture de code idempotent pour vos fonctions garantit ne gestion identique des événements dupliqués. Votre code doit valider correctement les événements et gérer correctement les événements dupliqués. Pour de plus amples informations, veuillez consulterComment faire en sorte que ma fonction Lambda soit idempotente ?.

> **Note**
>
> Vous pouvez utiliser Powertools pour AWS Lambda rendre les fonctions idempotentes. Pour en savoir plus, consultez :
>
> - Python - Utilitaire d'idempotence
> - TypeScript - Utilitaire d'impuissance
> - Java - Utilitaire d'idempotence
> - .NET - Utilitaire d'idempotence

Pour connaître les pratiques exemplaires en matière de code spécifiques à chaque langage, reportez-vous aux sections suivantes :

- Pratiques exemplaires en matière de code pour les fonctions Lambda Node.js
- Meilleures pratiques en matière de code pour les fonctions TypeScript Lambda
- Pratiques exemplaires en matière de code pour les fonctions Lambda Python

- Pratiques exemplaires en matière de code pour les fonctions Lambda Ruby
- Pratiques exemplaires en matière de code pour les fonctions Lambda Java
- Pratiques exemplaires en matière de code pour les fonctions Lambda Go
- Pratiques exemplaires de codage pour les fonctions Lambda C#
- Pratiques exemplaires en matière de code pour les fonctions Lambda Rust

Configuration de la fonction

Le test de performance de votre fonction Lambda est une partie cruciale pour garantir que vous choisissez la configuration de taille mémoire optimale. Toute augmentation de la taille mémoire déclenche une augmentation équivalente de l'UC disponible pour votre fonction. L'utilisation de la mémoire pour votre fonction est déterminée par appel et peut être consultée sur Amazon CloudWatch. À chaque invocation, une entrée `REPORT:` est créée, comme indiqué ci-dessous :

```
REPORT RequestId: 3604209a-e9a3-11e6-939a-754dd98c7be3 Duration: 12.34 ms Billed
 Duration: 100 ms Memory Size: 128 MB Max Memory Used: 18 MB
```

En analysant le champ `Max Memory Used:`, vous pouvez déterminer si votre fonction a besoin de plus de mémoire ou si vous avez surdimensionné la taille mémoire de votre fonction.

Pour trouver la configuration de mémoire adaptée à vos fonctions, nous vous recommandons d'utiliser le projet open source AWS Lambda Power Tuning. Pour plus d'informations, voir Réglage AWS Lambda de l'alimentation activé GitHub.

Pour optimiser les performances des fonctions, nous recommandons également de déployer des bibliothèques capables de tirer parti des extensions vectorielles avancées 2 (AVX2). Cela vous permet de traiter des charges de travail exigeantes, comme l'inférence du machine learning, le traitement multimédia, le calcul haute performance (HPC), les simulations scientifiques et la modélisation financière. Pour plus d'informations, consultez la section Création de AWS Lambda fonctions plus rapides avec AVX2.

Effectuez un test de charge de votre fonction Lambda pour déterminer une valeur de délai d'expiration optimale. Il importe d'analyser comment votre fonction s'exécute afin que vous puissiez mieux déterminer les problèmes de service de dépendance qui peuvent accroître la simultanéité de la fonction au-delà de ce que vous attendez. Cet aspect est particulièrement important quand votre fonction Lambda effectue des appels réseau aux ressources qui peuvent ne pas gérer la mise à

l'échelle de Lambda. Pour plus d'informations sur le test de charge de votre application, consultez Test de charge distribué sur AWS.

Utilisez les autorisations les plus restrictives lors de la définition des stratégies IAM. Maîtrisez les ressources et les opérations dont votre fonction Lambda a besoin et limitez le rôle d'exécution à ces autorisations. Pour plus d'informations, consultez Gestion des autorisations dans AWS Lambda.

Familiarisez-vous avec Quotas Lambda. La taille de la charge utile, les descripteurs de fichiers et l'espace /tmp sont souvent ignorés lors de la détermination des limites des ressources.

Supprimez les fonctions Lambda que vous n'utilisez plus. En procédant ainsi, les fonctions inutilisées n'interviendront plus inutilement dans la limite de taille de votre package de déploiement.

Si vous utilisez Amazon Simple Queue Service en tant que source d'événement, assurez-vous que la valeur de la durée de l'invocation prévue de la fonction ne dépasse pas la valeur Délai de visibilité de la file d'attente. Cela s'applique à CreateFunction et UpdateFunctionConfiguration.

- Dans le cas de CreateFunction, le processus de création de la fonction AWS Lambda échouera.
- Dans le cas de UpdateFunctionConfiguration, cela pourrait entraîner des appels dupliqués de la fonction.

Capacité de mise à l'échelle de la fonction

Familiarisez-vous avec vos contraintes de débit en amont et en aval. Bien que les fonctions Lambda se mettent à l'échelle parfaitement, les dépendances en amont et en aval peuvent avoir des mêmes capacités de débit différentes. Si vous devez limiter le niveau de mise à l'échelle de votre fonction, configurez la simultanéité réservée sur votre fonction.

Créez une tolérance à la limitation. Si votre fonction synchrone est ralentie en raison d'un trafic dépassant le taux de mise à l'échelle de Lambda, vous pouvez utiliser les stratégies suivantes pour améliorer la tolérance à la limitation :

- Utilisez les délais d'expiration, les nouvelles tentatives et le backoff avec instabilité. La mise en œuvre de ces stratégies facilite les nouvelles tentatives d'invocation et permet de garantir que Lambda peut se mettre à l'échelle en quelques secondes afin de minimiser la limitation des utilisateurs finaux.
- Utilisez la simultanéité allouée. La simultanéité allouée est le nombre d'environnements d'exécution préinitialisés que Lambda alloue à votre fonction. Lambda gère les requêtes entrantes en utilisant la simultanéité allouée lorsqu'elle est disponible. Lambda peut également mettre à l'échelle votre

fonction au-delà de votre paramètre de simultanéité allouée si nécessaire. La configuration de la simultanéité provisionnée entraîne des frais supplémentaires pour votre compte. AWS

Métriques et alarmes

Utilisez Utilisation des métriques CloudWatch avec Lambda les CloudWatch alarmes et au lieu de créer ou de mettre à jour une métrique à partir de votre code de fonction Lambda. C'est une manière beaucoup plus efficace de suivre l'état de vos fonctions Lambda, qui vous permet de détecter de façon précoce d'éventuels problèmes dans le processus de développement. Par exemple, vous pouvez configurer une alarme basée sur la durée attendue de l'invocation de votre fonction Lambda afin de pouvoir prendre en compte les goulots d'étranglement ou les latences du code de votre fonction.

Émettez des métriques personnalisées de manière asynchrone à l'aide du format EMF (Embedded Metric Format). Au lieu d'effectuer des appels d'API synchrones CloudWatch, utilisez EMF pour émettre des métriques via les journaux de votre fonction. Cette approche réduit la latence et améliore les performances. L'utilitaire de métriques de Powertools pour AWS Lambda gère automatiquement le formatage EMF. Pour plus d'informations, consultez les utilitaires Python TypeScript, Java ou .NET Metrics dans la AWS Lambda documentation Powertools. Pour plus d'informations sur l'utilisation d'EMF pour générer des journaux au format métrique, consultez la section Publication de journaux avec le format métrique intégré dans le guide de l' CloudWatch utilisateur Amazon.

Utilisez la journalisation JSON structurée pour une meilleure observabilité. La journalisation structurée facilite la recherche, le filtrage et l'analyse des journaux de votre fonction. Envisagez d'utiliser l'utilitaire Logger de Powertools pour AWS Lambda formater automatiquement les journaux au format JSON. Pour plus d'informations, consultez les utilitaires Python TypeScript, Java ou .NET Logger dans la documentation Powertools. AWS Lambda

Mettez à profit votre bibliothèque de journalisation et les dimensions et métriques AWS Lambda pour intercepter les erreurs d'application (par exemple, ERR, ERROR, WARNING, etc.)

Utiliser Détection des anomalies de coûts AWS pour détecter les activités inhabituelles sur votre compte. La détection des anomalies de coûts utilise le machine learning pour effectuer une surveillance permanent de votre coût et votre utilisation, tout en minimisant les alertes de faux positifs. La détection des anomalies de coût utilise les données provenant de AWS Cost Explorer, avec un délai pouvant aller jusqu'à 24 heures. Par conséquent, après utilisation, jusqu'à 24 heures peuvent être nécessaires pour détecter une anomalie. Afin de vous familiariser à la détection

des anomalies de coûts, vous devez d'abord vous inscrire à Cost Explorer. Ensuite, accéder à la détection des anomalies des coûts.

Utilisation des flux

Testez différentes tailles de lot et d'enregistrement de telle sorte que la fréquence d'interrogation de chaque source d'événement soit réglée sur la vitesse à laquelle votre fonction peut exécuter sa tâche. Le CreateEventSourceMapping BatchSize paramètre contrôle le nombre maximum d'enregistrements pouvant être envoyés à votre fonction à chaque appel. Une taille de lot plus grande peut souvent absorber plus efficacement l'invocation sur un plus large ensemble d'enregistrements, ce qui accroît votre débit.

Par défaut, Lambda invoque votre fonction dès que des enregistrements sont disponibles. Si le lot que Lambda lit à partir de la source d'événements ne comprend qu'un seul enregistrement, Lambda envoie un seul registre à la fonction. Pour éviter d'invoquer la fonction avec un petit nombre de registres, vous pouvez indiquer à la source d'événement de les mettre en mémoire tampon pendant 5 minutes en configurant une fenêtre de traitement par lots. Avant d'invoquer la fonction, Lambda continue de lire les registres de la source d'événements jusqu'à ce qu'il ait rassemblé un lot complet, que la fenêtre de traitement par lot expire ou que le lot atteigne la limite de charge utile de 6 Mo. Pour de plus amples informations, veuillez consulter Comportement de traitement par lots.

> **Warning**
>
> Les mappages des sources d'événements Lambda traitent chaque événement au moins une fois, et le traitement des enregistrements peut être dupliqué. Pour éviter les problèmes potentiels liés à des événements dupliqués, nous vous recommandons vivement de rendre votre code de fonction idempotent. Pour en savoir plus, consultez Comment rendre ma fonction Lambda idempotente dans le Knowledge Center. AWS

Activez la réponse partielle de lot pour le traitement des flux. Lorsque vous traitez des lots d'enregistrements provenant de flux comme Kinesis ou DynamoDB Streams, activez la réponse partielle de lot pour permettre à Lambda de réessayer uniquement les enregistrements ayant échoué au lieu de réessayer le lot complet. Cela améliore l'efficacité du traitement et réduit les retraitements inutiles. Vous pouvez éventuellement utiliser l'utilitaire Batch de Powertools AWS Lambda pour simplifier les modèles de traitement par lots.

> **Note**
>
> Vous pouvez utiliser Powertools AWS Lambda pour le traitement par lots. Pour en savoir plus, consultez :
>
> - Python - Traitement par lots
> - TypeScript - Traitement par lots
> - Java - Traitement par lots
> - .NET – Traitement par lots

Augmentez le débit de traitement du flux Kinesis en ajoutant des partitions. Un flux Kinesis est composé d'une ou plusieurs partitions. La vitesse à laquelle Lambda peut lire les données de Kinesis s'échelonne linéairement en fonction du nombre de partitions. L'augmentation du nombre de partitions entraîne directement celle du nombre maximum d'invocations simultanés de fonctions Lambda, et peut accroître le débit de traitement de votre flux Kinesis. Pour de plus amples informations sur la relation entre les partitions et les invocations de fonctions, consultez the section called "Flux d'interrogation et de mise en lots". Si vous augmentez le nombre de partitions dans un flux Kinesis, assurez-vous d'avoir choisi une clé de partition appropriée (consultez Clés de partition) pour vos données, de façon à ce que les enregistrements associés se retrouvent sur les mêmes partitions et que vos données soient correctement distribuées.

Utilisez Amazon CloudWatch on IteratorAge pour déterminer si votre flux Kinesis est en cours de traitement. Par exemple, configurez une CloudWatch alarme avec un réglage maximal de 30 000 (30 secondes).

Bonnes pratiques de sécurité

Surveillez votre utilisation AWS Lambda en ce qui concerne les meilleures pratiques de sécurité en utilisant AWS Security Hub CSPM. Security Hub CSPM utilise des contrôles de sécurité pour évaluer les configurations des ressources et les normes de sécurité afin de vous aider à vous conformer aux différents cadres de conformité. Pour plus d'informations sur l'utilisation de Security Hub CSPM pour évaluer les ressources Lambda, consultez la section AWS Lambda Contrôles du Guide de l'utilisateur. AWS Security Hub CSPM

Surveillez les journaux d'activité du réseau Lambda à l'aide d'Amazon Lambda Protection GuardDuty . GuardDuty La protection Lambda vous aide à identifier les menaces de sécurité

potentielles lorsque des fonctions Lambda sont invoquées dans votre. Compte AWS Par exemple, si l'une de vos fonctions interroge une adresse IP associée à une activité liée aux cryptomonnaies. GuardDuty surveille les journaux d'activité réseau générés lorsqu'une fonction Lambda est invoquée. Pour en savoir plus, consultez la section Protection Lambda dans le guide de GuardDuty l'utilisateur Amazon.

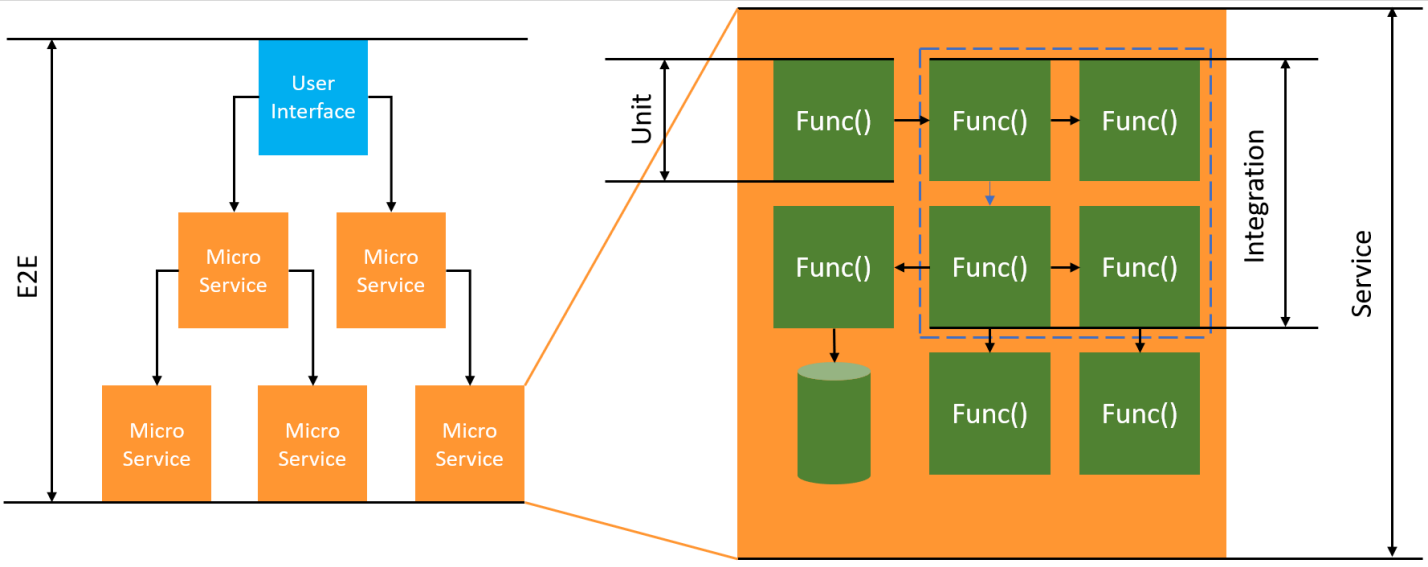
Commenter tester des fonctions et des applications sans serveur

Le test des fonctions sans serveur utilise les types et techniques de tests traditionnels, mais vous devez également envisager de tester les applications sans serveur dans leur ensemble. Les tests basés sur le cloud fourniront la mesure la plus précise de la qualité de vos fonctions et de vos applications sans serveur.

Une architecture d'application sans serveur comprend des services gérés qui fournissent des fonctionnalités d'applications critiques par le biais d'appels d'API. C'est pourquoi votre cycle de développement doit inclure des tests automatisés qui vérifient la fonctionnalité lorsque votre fonction et vos services interagissent.

Si vous ne créez pas de tests basés sur le cloud, vous pouvez rencontrer des problèmes en raison des différences entre votre environnement local et l'environnement déployé. Votre processus d'intégration continue doit exécuter des tests sur une série de ressources allouées dans le cloud avant de promouvoir votre code vers l'environnement de déploiement suivant, comme l'assurance qualité, la mise en place ou la production.

Poursuivez la lecture de ce petit guide pour en savoir plus sur les stratégies de test pour les applications sans serveur, ou rendez-vous sur le référentiel Serverless Test Samples pour vous plonger dans des exemples pratiques, spécifiques au langage et à l'exécution que vous avez choisis.



Pour les tests sans serveur, vous écrirez toujours des tests unitaires, d'intégration et de bout en bout.

- Tests unitaires : tests exécutés sur un bloc de code isolé. Par exemple, la vérification de la logique métier permettant de calculer les frais de livraison en fonction d'un article et d'une destination donnés.
- Tests d'intégration : tests impliquant au moins deux composants ou services qui interagissent, généralement dans un environnement cloud. Par exemple, la vérification d'une fonction traite les événements d'une file d'attente.
- Tests de bout en bout : tests qui vérifient le comportement de l'ensemble d'une application. Il s'agit par exemple de s'assurer que l'infrastructure est correctement mise en place et que les événements circulent entre les services comme prévu pour enregistrer la commande d'un client.

Résultats commerciaux ciblés

Le test des solutions sans serveur peut nécessiter un peu plus de temps pour mettre en place des tests qui vérifient les interactions dirigées par les événements entre les services. En lisant ce guide, gardez à l'esprit les raisons commerciales pratiques suivantes :

- Améliorer la qualité de votre application
- Réduire le temps nécessaire à la création de fonctionnalités et à la correction des bogues

La qualité d'une application dépend de la réalisation de tests dans différents scénarios afin de vérifier sa fonctionnalité. Une réflexion approfondie sur les scénarios commerciaux et l'automatisation de ces tests pour les exécuter sur des services cloud améliorera la qualité de votre application.

Les bogues logiciels et les problèmes de configuration ont le moins d'impact sur les coûts et le calendrier lorsqu'ils sont détectés au cours d'un cycle de développement itératif. Si des problèmes ne sont pas détectés au cours du développement, leur détection et leur résolution en production exigent davantage d'efforts de la part d'un plus grand nombre de personnes.

Une stratégie de test sans serveur bien planifiée augmentera la qualité des logiciels et améliorera le temps d'itération en vérifiant que vos fonctions Lambda et vos applications fonctionnent comme prévu dans un environnement cloud.

Que faut-il tester ?

Nous vous recommandons d'adopter une stratégie de test qui évalue les comportements des services gérés, la configuration du cloud, les politiques de sécurité et l'intégration à votre code afin d'améliorer la qualité du logiciel. Les tests de comportement, également connus sous le nom de tests

en boîte noire, permettent de vérifier qu'un système fonctionne comme prévu sans en connaître tous les rouages.

- Exécutez des tests unitaires pour vérifier la logique métier au sein des fonctions Lambda.
- Vérifiez que les services intégrés sont réellement invoqués et que les paramètres d'entrée sont corrects.
- Vérifiez qu'un événement passe en revue tous les services attendus de bout en bout dans un flux de travail.

Dans les architectures traditionnelles basées sur des serveurs, les équipes définissent souvent une portée de test afin d'inclure uniquement le code qui s'exécute sur le serveur d'application. Les autres composants, services ou dépendances sont souvent considérés comme externes et hors de portée des tests.

Les applications sans serveur sont souvent composées de petites unités de travail, telles que les fonctions Lambda qui récupèrent des produits d'une base de données, traitent des éléments d'une file d'attente ou redimensionnent une image stockée. Chaque composant s'exécute dans son propre environnement. Les équipes seront probablement responsables de plusieurs de ces petites unités au sein d'une même application.

Certaines fonctionnalités de l'application peuvent être entièrement déléguées à des services gérés tels qu'Amazon S3, ou créées sans utiliser de code développé en interne. Il n'est pas nécessaire de tester ces services gérés, mais vous devez tester l'intégration avec ces services.

Comment tester le sans serveur

Vous savez probablement comment tester des applications déployées localement : vous rédigez des tests qui s'exécutent sur du code fonctionnant entièrement sur votre système d'exploitation de bureau ou à l'intérieur de conteneurs. Par exemple, vous pouvez invoquer un composant de service Web local avec une requête, puis faire des assertions sur la réponse.

Les solutions sans serveur sont conçues à partir du code de votre fonction et de services gérés basés sur le cloud, tels que les files d'attente, les bases de données, les bus d'événements et les systèmes de messagerie. Ces composants sont tous connectés par le biais d'une architecture orientée événements, dans laquelle les messages, appelés événements, circulent d'une ressource à l'autre. Ces interactions peuvent être synchrones, par exemple lorsqu'un service Web renvoie des résultats immédiatement, ou une action asynchrone qui se termine ultérieurement, comme le

placement d'éléments dans une file d'attente ou le démarrage d'une étape d'un flux de travail. Votre stratégie de test doit inclure les deux scénarios et tester les interactions entre les services. Pour les interactions asynchrones, il peut être nécessaire de détecter des effets secondaires dans les composants en aval qui peuvent ne pas être immédiatement observables.

Il n'est pas pratique de répliquer l'ensemble d'un environnement cloud, y compris les files d'attente, les tables de base de données, les bus d'événements, les politiques de sécurité, etc. Vous rencontrerez inévitablement des problèmes en raison des différences entre votre environnement local et vos environnements déployés dans le cloud. Les variations entre vos environnements augmenteront le temps nécessaire à la reproduction et à la correction des bogues.

Dans les applications sans serveur, les composants de l'architecture existent généralement à part entière dans le cloud. Il est donc nécessaire de les tester par rapport au code et aux services du cloud pour développer des fonctionnalités et corriger les bogues.

Techniques de test

En réalité, votre stratégie de test comprendra probablement une combinaison de techniques visant à améliorer la qualité de vos solutions. Vous utiliserez des tests interactifs rapides pour déboguer les fonctions de la console, des tests unitaires automatisés pour vérifier la logique métier isolée, la vérification des appels vers des services externes à l'aide de simulations, et des tests occasionnels sur des émulateurs imitant un service.

- the section called "Tests dans le cloud" : vous déployez l'infrastructure et le code à tester avec des services réels, des politiques de sécurité, des configurations et des paramètres spécifiques à l'infrastructure. Les tests basés sur le cloud fournissent la mesure la plus précise de la qualité de votre code.

  Le débogage d'une fonction dans la console est un moyen rapide de la tester dans le cloud. Vous pouvez choisir parmi une bibliothèque d'exemples d'événements de test ou créer un événement personnalisé pour tester une fonction de manière isolée. Vous pouvez également partager des événements de test via la console avec votre équipe.

  Pour automatiser les tests au cours du cycle de vie du développement et de génération, vous devez effectuer des tests en dehors de la console. Consultez les sections de ce guide relatives aux tests spécifiques à chaque langue pour connaître les stratégies et les ressources d'automatisation.
- the section called "Tester avec des simulations" : les simulations sont des objets de votre code qui simulent et remplacent un service externe. Les simulations fournissent un comportement prédéfini

pour vérifier les appels de service et les paramètres. Un faux est une implémentation fictive qui utilise des raccourcis pour simplifier ou améliorer les performances. Par exemple, un faux objet d'accès aux données peut renvoyer des données depuis un entrepôt de données en mémoire. Les simulations peuvent imiter et simplifier des dépendances complexes, mais peuvent également conduire à d'autres simulations afin de remplacer les dépendances imbriquées.

- the section called "Tester localement en utilisant AWS SAM CLI" : utilisez AWS SAM CLI pour invoquer localement des fonctions Lambda dans des conteneurs Docker qui utilisent le même environnement d'exécution qu'AWS Lambda. Vous pouvez tester la logique d'une fonction et le traitement des événements sans déploiement dans le cloud.
- the section called "Tester avec émulation" : utilisez l'intégration LocalStack dans VS Code pour émuler plusieurs services AWS localement afin de tester les intégrations de service.

Tests dans le cloud

Les tests dans le cloud sont utiles pour toutes les phases des tests, y compris les tests unitaires, les tests d'intégration et les tests de bout en bout. Lorsque vous exécutez des tests sur du code basé sur le cloud qui interagit également avec des services basés sur le cloud, vous obtenez la mesure la plus précise de la qualité de votre code.

Un moyen pratique d'exécuter une fonction Lambda dans le cloud consiste à utiliser un événement de test dans la AWS Management Console. Un événement de test est une entrée JSON pour votre fonction. Si votre fonction ne nécessite pas d'entrée, l'événement peut être un document JSON vide (`{}`). La console fournit des exemples d'événements pour diverses intégrations de services. Après avoir créé un événement dans la console, vous pouvez également le partager avec votre équipe pour faciliter les tests et les rendre plus cohérents.

Découvrez comment déboguer un exemple de fonction dans la console.

> **Note**
>
> Bien que l'exécution de fonctions dans la console soit un moyen rapide de déboguer, l'automatisation de vos cycles de test est essentielle pour améliorer la qualité des applications et la vitesse de développement.

Des exemples d'automatisation des tests sont disponibles dans le référentiel Serverless Test Samples. La ligne de commande suivante exécute un exemple de test d'intégration Python automatique :

```
python -m pytest -s tests/integration -v
```

Bien que le test soit exécuté localement, il interagit avec des ressources basées dans le cloud. Ces ressources ont été déployées en utilisant le AWS Serverless Application Model et l'outil de commande en ligne AWS SAM. Le code de test récupère d'abord les sorties de la pile déployée, qui comprend le point de terminaison de l'API, l'ARN de la fonction et le rôle de sécurité. Le test envoie ensuite une demande au point de terminaison de l'API, qui répond par une liste de compartiments Amazon S3. Ce test s'exécute entièrement sur des ressources basées sur le cloud afin de vérifier que ces ressources sont déployées, sécurisées et qu'elles fonctionnent comme prévu.

```
========================= test session starts ==========================
      platform darwin -- Python 3.10.10, pytest-7.3.1, pluggy-1.0.0
      -- /Users/t/code/aws/serverless-test-samples/python-test-samples/apigw-lambda/
venv/bin/python
      cachedir: .pytest_cache
      rootdir: /Users/t/code/aws/serverless-test-samples/python-test-samples/apigw-
lambda
      plugins: mock-3.10.0
      collected 1 item


      tests/integration/test_api_gateway.py::TestApiGateway::test_api_gateway

      --> Stack outputs:

        HelloWorldApi
        = https://p7teqs3162.execute-api.us-east-2.amazonaws.com/Prod/hello/
        > API Gateway endpoint URL for Prod stage for Hello World function

        PythonTestDemo
        = arn:aws:lambda:us-east-2:123456789012:function:testing-apigw-lambda-
PythonTestDemo-iSij8evaTdxl
        > Hello World Lambda Function ARN

        PythonTestDemoIamRole
        = arn:aws:iam::123456789012:role/testing-apigw-lambda-PythonTestDemoRole-
IZELQQ9MG4HQ
        > Implicit IAM Role created for Hello World function

      --> Found API endpoint for "testing-apigw-lambda" stack...
      --> https://p7teqs3162.execute-api.us-east-2.amazonaws.com/Prod/hello/
      API Gateway response:
```

```
      amplify-dev-123456789-deployment|myapp-prod-p-loggingbucket-123456|s3-java-
bucket-123456789
      PASSED

      ========================= 1 passed in 1.53s =========================
```

Pour le développement d'applications natives cloud, les tests dans le cloud offrent les avantages suivants :

- Vous pouvez tester tous les services disponibles.
- Vous utilisez toujours les API de service et les valeurs de retour les plus récentes.
- Un environnement de test dans le cloud ressemble beaucoup à votre environnement de production.
- Les tests peuvent porter sur les politiques de sécurité, les quotas de service, les configurations et les paramètres spécifiques à l'infrastructure.
- Chaque développeur peut rapidement créer un ou plusieurs environnements de test dans le cloud.
- Les tests dans le cloud augmentent la confiance dans le fait que votre code fonctionnera correctement en production.

Les tests dans le cloud présentent certains inconvénients. L'inconvénient le plus évident des tests dans le cloud est que les déploiements vers des environnements cloud prennent généralement plus de temps que les déploiements vers des environnements de bureau locaux.

Heureusement, des outils tels que AWS Serverless Application Model (AWS SAM) Accelerate, le mode de surveillance du Cloud Development Kit AWS (AWS CDK) et le SST (tiers) réduisent le temps de latence lié aux itérations de déploiement dans le cloud. Ces outils peuvent surveiller votre infrastructure et votre code et déploient automatiquement des mises à jour incrémentielles dans votre environnement cloud.

> **Note**
>
> Découvrez comment créer une infrastructure en tant que code dans le Guide du développeur sans serveur pour en savoir plus sur AWS Serverless Application Model, CloudFormation, et AWS Cloud Development Kit (AWS CDK).

Contrairement aux tests locaux, les tests dans le cloud nécessitent des ressources supplémentaires qui peuvent entraîner des coûts de service. La création d'environnements de test isolés peut

accroître la charge de travail de vos équipes DevOps, en particulier dans les organisations où les comptes et l'infrastructure font l'objet de contrôles stricts. Néanmoins, lorsque l'on travaille avec des scénarios d'infrastructure complexes, le coût en temps de développement pour mettre en place et maintenir un environnement local complexe peut être similaire (voire plus coûteux) que l'utilisation d'environnements de test jetables créés avec des outils d'automatisation d'infrastructure en tant que code.

Même en tenant compte de ces considérations, les tests dans le cloud restent le meilleur moyen de garantir la qualité de vos solutions sans serveur.

Tester avec des simulations

Les tests avec des simulations sont une technique qui consiste à créer des objets de remplacement dans votre code afin de simuler le comportement d'un service cloud.

Par exemple, vous pouvez écrire un test qui utilise une simulation du service Amazon S3 qui renvoie une réponse spécifique chaque fois que la méthode CreateObject est appelée. Lorsqu'un test s'exécute, la simulation renvoie cette réponse programmée sans appeler Amazon S3 ni aucun autre point de terminaison de service.

Les objets simulés sont souvent générés par un cadre simulé afin de réduire les efforts de développement. Certains cadres simulés sont génériques tandis que d'autres sont conçus spécifiquement pour les kits AWS SDK, tels que Moto, une bibliothèque Python pour simuler des services et des ressources AWS.

Il convient de noter que les objets simulés diffèrent des émulateurs dans la mesure où les simulations sont généralement créées ou configurées par un développeur dans le cadre du code de test, alors que les émulateurs sont des applications autonomes qui exposent les fonctionnalités de la même manière que les systèmes qu'ils émulent.

Les avantages de l'utilisation de simulations incluent les avantages suivants :

- Les simulations peuvent simuler des services tiers qui échappent au contrôle de votre application, tels que les API et les fournisseurs de logiciels en tant que service (SaaS), sans qu'il soit nécessaire d'accéder directement à ces services.
- Les simulations sont utiles pour tester les conditions d'échec, en particulier lorsque ces conditions sont difficiles à simuler, comme dans le cas d'une panne de service.
- Une fois configurée, la simulation permet d'effectuer des tests locaux rapides.

- Les simulations peuvent fournir un comportement de substitution pour pratiquement n'importe quel type d'objet. Les stratégies de simulations peuvent donc couvrir une plus grande variété de services que les émulateurs.
- Lorsque de nouvelles fonctionnalités ou de nouveaux comportements sont disponibles, les tests simulés permettent de réagir plus rapidement. En utilisant un cadre générique de simulation, vous pouvez simuler de nouvelles fonctionnalités dès que le kit SDK AWS mis à jour est disponible.

Les tests simulés présentent les inconvénients suivants :

- Les simulations nécessitent généralement un effort d'installation et de configuration non négligeable, en particulier lorsqu'il s'agit de déterminer les valeurs de retour de différents services afin de simuler correctement les réponses.
- Les simulations sont écrites, configurées et doivent être mises à jour par les développeurs, ce qui accroît leurs responsabilités.
- Vous pouvez avoir besoin d'un accès au cloud pour comprendre les API et les valeurs de retour des services.
- Les simulations peuvent être difficiles à entretenir. Lorsque les signatures des API cloud simulées changent ou que les schémas de valeurs de retour évoluent, vous devez mettre à jour vos simulations. Les simulations nécessitent également des mises à jour si vous étendez la logique de votre application pour effectuer des appels vers de nouvelles API.
- Les tests utilisant des simulations peuvent réussir dans les environnements de bureau, mais échouer dans le cloud. Les résultats peuvent ne pas correspondre à l'API actuelle. La configuration du service et les quotas ne peuvent pas être testés.
- Les cadres simulés sont limités lorsqu'il s'agit de tester ou de détecter les politiques AWS Identity and Access Management (IAM) ou les limites de quota. Bien que les simulations permettent de mieux simuler l'échec d'une autorisation ou le dépassement d'un quota, les tests ne peuvent pas déterminer le résultat qui se produira réellement dans un environnement de production.

Tester localement en utilisant AWS SAM CLI

Utilisez AWS SAM CLI pour tester vos fonctions dans des conteneurs Docker en utilisant le même environnement d'exécution que AWS Lambda. Vous pouvez tester la logique d'une fonction et le traitement des événements localement sans déploiement dans le cloud. Si votre fonction fait des appels d'API à d'autres Services AWS, ces appels atteindront des ressources AWS réelles.

Les avantages des tests avec des conteneurs locaux incluent les suivants :

- Utilise des environnements d'exécution AWS Lambda pour des tests précis.
- Permet d'effectuer des itérations de développement local rapides sans déploiement dans le cloud.
- Prend en charge le débogage à l'aide d'outils de développement locaux familiers.

Les tests avec les conteneurs locaux présentent les limites suivantes :

- Les appels Service AWS de votre fonction interagiront avec des ressources AWS réelles, ce qui peut entraîner des coûts et affecter les données de production.
- Nécessite que Docker soit installé et exécuté localement.

Tester avec émulation

Les émulateurs sont des applications exécutées localement qui imitent les services AWS en fournissant des API et des valeurs de retour similaires. LocalStack est un outil d'émulation populaire qui fournit un environnement de développement local complet pour tester les intégrations de service.

LocalStack est un émulateur de Cloud AWS que vous pouvez utiliser pour tester des applications sans serveur localement. Vous pouvez tester les fonctions Lambda qui s'intègrent à des services comme DynamoDB, Amazon S3 et Amazon SQS sans vous connecter à des services AWS réels. Vous pouvez utiliser LocalStack dans AWS Toolkit pour VS Code.

Les avantages des tests avec émulateurs sont les suivants :

- Les émulateurs peuvent faciliter les itérations de développement et les tests locaux rapides.
- Les émulateurs fournissent un environnement familier aux développeurs habitués à développer du code dans un environnement local. Par exemple, si vous êtes habitué au développement d'une application à plusieurs niveaux, vous pouvez disposer d'un moteur de base de données et d'un serveur Web, similaires à ceux qui fonctionnent en production, exécutés sur votre ordinateur local afin de fournir une capacité de test rapide, locale et isolée.
- Les émulateurs ne nécessitent aucune modification de l'infrastructure cloud (telle que les comptes cloud des développeurs). Ils sont donc faciles à implémenter avec les modèles de test existants.
- Comme les émulateurs n'utilisent pas de ressources AWS réelles, vous n'aurez pas à payer de frais imprévus lorsque vous démarrez plusieurs services ou si vous laissez certaines ressources fonctionner pendant de longues périodes.

Les tests avec des émulateurs présentent les inconvénients suivants :

- Les émulateurs peuvent être difficiles à mettre en place et à répliquer, en particulier lorsqu'ils sont utilisés dans des pipelines CI/CD. Cela peut accroître la charge de travail du personnel informatique ou des développeurs qui gèrent leurs propres logiciels.
- Les fonctionnalités et les API émulées sont généralement en retard par rapport aux mises à jour des services. Cela peut entraîner des erreurs parce que le code testé ne correspond pas à l'API réelle, et entraver l'adoption de nouvelles fonctionnalités.
- Les émulateurs nécessitent une assistance, des mises à jour, des corrections de bogues et des améliorations de la parité des fonctions. La responsabilité en incombe à l'auteur de l'émulateur, qui peut être une société tierce.
- Les tests qui s'appuient sur des émulateurs peuvent donner des résultats positifs au niveau local, mais échouer dans le cloud en raison des politiques de sécurité de la production, des configurations interservices ou du dépassement des quotas Lambda.

Bonnes pratiques

Les sections suivantes fournissent des recommandations pour réussir les tests d'applications sans serveur.

Vous trouverez des exemples pratiques de tests et d'automatisation des tests dans le référentiel Serverless Test Samples.

Prioriser les tests dans le cloud

Les tests dans le cloud fournissent la couverture de test la plus fiable, la plus précise et la plus complète. La réalisation de tests dans le contexte du cloud permettra de tester de manière exhaustive non seulement la logique métier, mais également les politiques de sécurité, les configurations de service, les quotas, ainsi que les signatures d'API et les valeurs de retour les plus récentes.

Structurer votre code pour le rendre testable

Simplifiez vos tests et vos fonctions Lambda en séparant le code spécifique à Lambda de votre logique métier principale.

Votre gestionnaire de fonctions Lambda doit être un adaptateur léger qui prend en charge les données des événements et ne transmet que les détails importants à votre ou vos méthodes de logique métier. Avec cette stratégie, vous pouvez envelopper des tests complets autour de votre logique métier sans vous soucier des détails spécifiques à Lambda. Vos fonctions Lambda AWS ne

doivent pas nécessiter la mise en place d'un environnement complexe ou d'une grande quantité de dépendances pour créer et initialiser le composant testé.

D'une manière générale, vous devez écrire un gestionnaire qui extrait et valide les données des objets d'événement et de contexte entrants, puis envoie ces données aux méthodes qui exécutent votre logique métier.

Accélérer les boucles de rétroaction du développement

Il existe des outils et des techniques permettant d'accélérer les boucles de rétroaction du développement. Par exemple, AWS SAM Accelerate et le mode de surveillance AWS CDK réduisent tous deux le temps nécessaire à la mise à jour des environnements cloud.

Les exemples du référentiel GitHub Serverless Test Samples explorent certaines de ces techniques.

Nous vous recommandons également de créer et de tester les ressources cloud le plus tôt possible au cours du développement, et pas seulement après une vérification du contrôle du code source. Cette pratique permet d'accélérer l'exploration et l'expérimentation lors du développement de solutions. En outre, l'automatisation du déploiement à partir d'une machine de développement vous permet de découvrir plus rapidement les problèmes de configuration du cloud et de réduire les efforts inutiles liés aux mises à jour et aux processus de révision du code.

Se concentrer sur les tests d'intégration

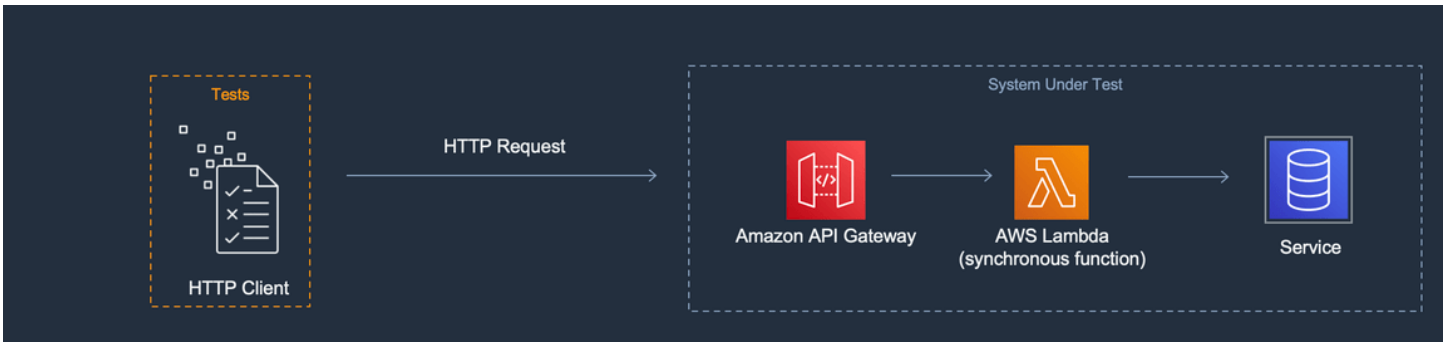
Lors de la création d'applications avec Lambda, il est recommandé de tester les composants ensemble.

Les tests exécutés sur deux composants architecturaux ou plus sont appelés tests d'intégration. L'objectif des tests d'intégration est de comprendre non seulement comment votre code s'exécutera entre les composants, mais aussi comment l'environnement qui héberge votre code se comportera. Les tests de bout en bout sont des types de tests d'intégration particuliers qui vérifient les comportements dans l'ensemble d'une application.

Pour créer des tests d'intégration, déployez votre application dans un environnement cloud. Cela peut se faire à partir d'un environnement local ou par le biais d'un pipeline CI/CD. Rédigez ensuite des tests pour exécuter le système sous test (SUT) et valider le comportement attendu.

Par exemple, le système testé pourrait être une application qui utilise API Gateway, Lambda et DynamoDB. Un test peut effectuer un appel HTTP synthétique vers un point de terminaison d'API

Gateway et valider que la réponse contient la charge utile attendue. Ce test valide que le code AWS Lambda est correct et que chaque service est correctement configuré pour gérer la demande, y compris les autorisations IAM entre eux. En outre, vous pouvez concevoir le test pour écrire des enregistrements de différentes tailles afin de vérifier que vos quotas de service, tels que la taille d'enregistrement maximale dans DynamoDB, sont correctement configurés.



Créer des environnements de test isolés

Les tests dans le cloud nécessitent généralement des environnements de développement isolés, de sorte que les tests, les données et les événements ne se chevauchent pas.

L'une des approches consiste à fournir à chaque développeur un compte AWS dédié. Cela permettra d'éviter les conflits de dénomination des ressources qui peuvent survenir lorsque plusieurs développeurs travaillant dans une base de code partagée tentent de déployer des ressources ou d'invoquer une API.

Les processus de test automatisés doivent créer des ressources portant un nom unique pour chaque pile. Par exemple, vous pouvez configurer des scripts ou des fichiers de configuration TOML de sorte que les commandes AWS SAM CLI sam deploy ou sam sync spécifient automatiquement une pile avec un préfixe unique.

Dans certains cas, les développeurs partagent un compte AWS. Cela peut être dû au fait que les ressources de votre pile sont coûteuses à exploiter, ou à provisionner et à configurer. Par exemple, une base de données peut être partagée pour faciliter la configuration et remplir correctement les données

Si les développeurs partagent un compte, vous devez définir des limites afin d'identifier les propriétaires et d'éliminer les chevauchements. L'un des moyens d'y parvenir est de mettre des préfixes sur les noms de piles avec les identifiants d'utilisateurs des développeurs. Une autre approche populaire consiste à configurer des piles basées sur des branches de code. Avec les limites de branches, les environnements sont isolés, mais les développeurs peuvent toujours partager des

ressources, telles qu'une base de données relationnelle. Cette approche est une bonne pratique lorsque les développeurs travaillent sur plusieurs branches à la fois.

Les tests dans le cloud sont utiles pour toutes les phases des tests, y compris les tests unitaires, les tests d'intégration et les tests de bout en bout. Il est essentiel de maintenir une isolation adéquate, mais vous souhaitez tout de même que votre environnement d'assurance qualité ressemble le plus possible à votre environnement de production. C'est pourquoi les équipes ajoutent des processus de contrôle des modifications pour les environnements d'assurance qualité.

Pour les environnements de pré-production et de production, les limites sont généralement définies au niveau du compte afin d'isoler les charges de travail de leurs bruyants voisins et de mettre en œuvre des contrôles de sécurité du moindre privilège pour protéger les données sensibles. Les charges de travail sont soumises à des quotas. Vous ne voulez pas que vos tests consomment les quotas alloués à la production (voisin bruyant) ou qu'ils aient accès aux données des clients. Les tests de charge constituent une autre activité que vous devez isoler de votre pile de production.

Dans tous les cas, les environnements doivent être configurés avec des alertes et des contrôles afin d'éviter des dépenses inutiles. Par exemple, vous pouvez limiter le type, le niveau ou la taille des ressources qui peuvent être créées, et configurer des alertes par e-mail lorsque les coûts estimés dépassent un seuil donné.

Utiliser des simulations pour isoler la logique métier

Les cadres simulés sont un outil précieux pour écrire des tests unitaires rapides. Ils sont particulièrement utiles lorsque les tests couvrent une logique interne complexe, tels que des calculs mathématiques ou financiers ou des simulations. Recherchez les tests unitaires qui comportent un grand nombre de cas de test ou de variations d'entrée, dans lesquels ces entrées ne modifient ni le modèle ni le contenu des appels vers d'autres services cloud.

Le code couvert par des tests unitaires avec des simulations doit également être couvert par des tests dans le cloud. Cela est recommandé, car un ordinateur portable de développeur ou un environnement de machine de génération peuvent être configurés différemment d'un environnement de production dans le cloud. Par exemple, vos fonctions Lambda peuvent utiliser plus de mémoire ou de temps que ce qui est alloué lorsqu'elles sont exécutées avec certains paramètres d'entrée. Votre code peut également inclure des variables d'environnement qui ne sont pas configurées de la même manière (ou pas du tout), et les différences peuvent entraîner un comportement différent ou un échec du code.

L'avantage des simulations est moindre pour les tests d'intégration, car le niveau d'effort pour mettre en œuvre les simulations nécessaires augmente avec le nombre de points de connexion. Les tests de bout en bout ne doivent pas utiliser de simulations, car ces tests traitent généralement d'états et de logiques complexes qui ne peuvent pas être facilement simulés avec des cadres simulés.

Enfin, évitez d'utiliser des services cloud simulés pour valider la bonne mise en œuvre des appels de service. Au lieu de cela, faites des appels de services dans le cloud pour valider le comportement, la configuration et la mise en œuvre fonctionnelle.

Utiliser les émulateurs avec parcimonie

Les émulateurs peuvent être pratiques dans certains cas d'utilisation, par exemple pour une équipe de développement disposant d'un accès Internet limité, peu fiable ou lent. Mais, dans la plupart des cas, choisissez d'utiliser les émulateurs avec parcimonie.

En évitant les émulateurs, vous serez en mesure de construire et d'innover avec les dernières fonctionnalités de service et les API les plus récentes. Vous ne serez pas obligé d'attendre les versions des fournisseurs pour atteindre la parité des fonctionnalités. Vous réduirez vos dépenses initiales et continues liées à l'achat et à la configuration sur plusieurs systèmes de développement et machines de génération. En outre, vous éviterez le problème lié au fait que de nombreux services cloud n'ont tout simplement pas d'émulateurs disponibles. Une stratégie de test qui repose sur l'émulation rendra impossible l'utilisation de ces services (ce qui entraînera des solutions de contournement potentiellement plus coûteuses) ou produira un code et des configurations qui ne sont pas correctement testés.

Lorsque vous utilisez l'émulation à des fins de test, vous devez tout de même effectuer des tests dans le cloud pour vérifier la configuration et tester les interactions avec les services cloud qui ne peuvent être simulés que dans un environnement émulé.

Défis liés aux tests locaux

Lorsque vous utilisez des émulateurs et des appels simulés pour tester sur votre bureau local, vous pouvez rencontrer des incohérences de test lorsque votre code progresse d'un environnement à l'autre dans votre pipeline CI/CD. Les tests unitaires visant à valider la logique métier de votre application sur votre bureau peuvent ne pas tester avec précision certains aspects critiques des services cloud.

Les exemples suivants présentent des cas à surveiller lors de tests locaux à l'aide de simulations et d'émulateurs :

Exemple : la fonction Lambda crée un compartiment S3

Si la logique d'une fonction Lambda dépend de la création d'un compartiment S3, un test complet doit confirmer qu'Amazon S3 a été appelé et que le compartiment a été créé avec succès.

- Dans le cas d'un test simulé, vous pouvez simuler une réponse de réussite et éventuellement ajouter un scénario de test pour gérer une réponse d'échec.
- Dans un scénario de test d'émulation, l'API CreateBucket peut être appelée, mais vous devez savoir que l'identité à l'origine de l'appel local ne proviendra pas du service Lambda. L'identité appelante n'assumera pas un rôle de sécurité comme elle le ferait dans le cloud. C'est pourquoi une authentification par espace réservé sera utilisée à la place, éventuellement avec un rôle ou une identité d'utilisateur plus permissif qui sera différente lorsqu'elle sera exécutée dans le cloud.

Les configurations de simulation et d'émulation testeront ce que fera la fonction Lambda si elle appelle Amazon S3 ; toutefois, ces tests ne vérifieront pas que la fonction Lambda, telle que configurée, est capable de créer correctement le compartiment Amazon S3. Vous devez vous assurer que le rôle attribué à la fonction dispose d'une politique de sécurité attachée qui autorise la fonction à effectuer l'action `s3:CreateBucket`. Si ce n'est pas le cas, la fonction échouera probablement lorsqu'elle sera déployée dans un environnement en nuage.

Exemple : la fonction Lambda traite les messages d'une file d'attente Amazon SQS

Si une file d'attente Amazon SQS est la source d'une fonction Lambda, un test complet doit vérifier que la fonction Lambda est correctement invoquée lorsqu'un message est placé dans une file d'attente.

Les tests d'émulation et les tests simulés sont généralement configurés pour exécuter directement le code de la fonction Lambda et pour simuler l'intégration d'Amazon SQS en transmettant une charge utile d'événement JSON (ou un objet désérialisé) en tant qu'entrée du gestionnaire de fonction.

Les tests locaux qui simulent l'intégration d'Amazon SQS testeront ce que la fonction Lambda fera lorsqu'elle sera invoquée par Amazon SQS avec une charge utile donnée, mais le test ne vérifiera pas qu'Amazon SQS invoquera avec succès la fonction Lambda lors de son déploiement dans un environnement cloud.

Voici quelques exemples de problèmes de configuration que vous pouvez rencontrer avec Amazon SQS et Lambda :

- Le délai de visibilité d'Amazon SQS est trop court, ce qui entraîne des invocations multiples alors qu'une seule était prévue.
- Le rôle d'exécution de la fonction Lambda ne permet pas de lire les messages de la file d'attente (via `sqs:ReceiveMessage`, `sqs:DeleteMessage`, ou `sqs:GetQueueAttributes`).
- L'exemple d'événement transmis à la fonction Lambda dépasse le quota de taille de message Amazon SQS. Par conséquent, le test n'est pas valide, car Amazon SQS ne sera jamais en mesure d'envoyer un message de cette taille.

Comme le montrent ces exemples, les tests qui couvrent la logique métier mais pas les configurations entre les services cloud sont susceptibles de fournir des résultats peu fiables.

FAQ

J'ai une fonction Lambda qui effectue des calculs et renvoie un résultat sans appeler aucun autre service. Dois-je vraiment le tester dans le cloud?

Oui. Les fonctions Lambda comportent des paramètres de configuration susceptibles de modifier le résultat du test. Tout le code de la fonction Lambda dépend des paramètres de délai d'attente et de mémoire, ce qui peut entraîner l'échec de la fonction si ces paramètres ne sont pas définis correctement. Les politiques Lambda permettent également la journalisation standard des sorties vers Amazon CloudWatch. Même si votre code n'appelle pas directement CloudWatch, une autorisation est requise pour activer la journalisation. Cette autorisation requise ne peut pas être simulée ou émulée avec précision.

Comment les tests dans le cloud peuvent-ils faciliter les tests unitaires? S'il se trouve dans le cloud et se connecte à d'autres ressources, ne s'agit-il pas d'un test d'intégration?

Nous définissons les tests unitaires comme des tests qui opèrent sur des composants architecturaux de manière isolée, ce qui n'empêche pas les tests d'inclure des composants susceptibles d'appeler d'autres services ou d'utiliser certaines communications réseau.

De nombreuses applications sans serveur possèdent des composants architecturaux qui peuvent être testés de manière isolée, même dans le cloud. Un exemple est celui d'une fonction Lambda qui prend une entrée, traite les données et envoie un message à une file d'attente Amazon SQS. Un test unitaire de cette fonction devrait permettre de vérifier si les valeurs d'entrée entraînent la présence de certaines valeurs dans le message en file d'attente.

Prenons l'exemple d'un test écrit en utilisant le modèle organiser, agir, affirmer :

- Organiser : allouez des ressources (une file d'attente pour recevoir des messages et la fonction en cours de test).
- Agir : appelez la fonction en cours de test.
- Affirmer : récupérez le message envoyé par la fonction et validez la sortie.

Une approche de test simulé consiste à simuler la file d'attente avec un objet simulé en cours de traitement et à créer une instance en cours de traitement de la classe ou du module contenant le code de la fonction Lambda. Pendant la phase d'assertion, le message en file d'attente est récupéré dans l'objet simulé.

Dans une approche basée sur le cloud, le test crée une file d'attente Amazon SQS pour les besoins du test et déploie la fonction Lambda avec des variables d'environnement configurées pour utiliser la file d'attente Amazon SQS isolée comme destination de sortie. Après avoir exécuté la fonction Lambda, le test récupère le message dans la file d'attente Amazon SQS.

Le test basé sur le cloud exécute le même code, confirme le même comportement et valide l'exactitude fonctionnelle de l'application. Cela aurait toutefois l'avantage supplémentaire de pouvoir valider les paramètres de la fonction Lambda : le rôle IAM, les politiques IAM, ainsi que les paramètres de délai d'expiration et de mémoire de la fonction.

Prochaines étapes et ressources

Les ressources suivantes vous permettront d'en savoir plus et de découvrir des exemples pratiques de tests.

Exemples d'implémentations

Le référentiel Serverless Test Samples sur GitHub contient des exemples concrets de tests qui suivent les modèles et les bonnes pratiques décrits dans ce guide. Le référentiel contient des exemples de code et des descriptions guidées des processus de simulation, d'émulation et de test dans le cloud décrits dans les sections précédentes. Utilisez ce référentiel pour vous mettre à jour sur les derniers conseils en matière de tests sans serveur d'AWS.

Suggestions de lecture

Rendez-vous sur Serverless Land pour accéder aux derniers blogs, vidéos et formations sur les technologies sans serveur d'AWS.

Il est également recommandé de lire les articles du blog AWS suivants :

- Accelerating serverless development with AWS SAM Accelerate (article du blog AWS)
- Increasing development speed with CDK Watch (article du blog AWS)
- Mocking service integrations with AWS Step Functions Local (article du blog AWS)
- Getting started with testing serverless applications (article du blog AWS)

Outils

- AWS SAM – test et débogage d'applications sans serveur
- AWS SAM – intégration avec des tests automatisés
- Lambda : test des fonctions Lambda dans la console Lambda

Améliorer les performances de démarrage avec Lambda SnapStart

Lambda SnapStart peut fournir des performances de démarrage inférieures à une seconde, généralement sans modification de votre code de fonction. SnapStart facilite la création d'applications hautement réactives et évolutives sans provisionnement de ressources ni mise en œuvre d'optimisations complexes des performances.

Le principal facteur de latence au démarrage (souvent appelé temps de démarrage à froid) est le temps que Lambda passe à initialiser la fonction, ce qui inclut le chargement du code de la fonction, le démarrage de l'environnement d'exécution et l'initialisation du code de la fonction. Avec SnapStart, Lambda initialise votre fonction lorsque vous publiez une version de fonction. Lambda prend un instantané Firecracker microVM de l'état de la mémoire et du disque de l'environnement d'exécution initialisé, chiffre l'instantané et le met intelligemment en cache pour optimiser la latence de récupération.

Pour garantir la résilience, Lambda conserve plusieurs copies de chaque instantané. Lambda corrige automatiquement les instantanés et leurs copies avec les dernières mises à jour de sécurité et de l'environnement d'exécution. Lorsque vous invoquez la version de la fonction pour la première fois, et au fur et à mesure que les invocations augmentent, Lambda reprend les nouveaux environnements d'exécution à partir de l'instantané mis en cache au lieu de les initialiser à partir de zéro, ce qui améliore la latence au démarrage.

> **Important**
>
> Si vos applications dépendent de l'unicité de l'état, vous devez évaluer votre code de fonction et vérifier qu'il est résilient aux opérations instantanées. Pour de plus amples informations, veuillez consulter Gestion de l'unicité avec Lambda SnapStart.

Rubriques

- Quand utiliser SnapStart
- Fonctions prises en charge et limitations
- Régions prises en charge
- Considérations de compatibilité
- SnapStart tarification

- Activation et gestion de Lambda SnapStart
- Gestion de l’unicité avec Lambda SnapStart
- Implémentation du code avant ou après les instantanés de fonctions Lambda
- Surveillance pour Lambda SnapStart
- Modèle de sécurité pour Lambda SnapStart
- Optimisation des performances Lambda SnapStart
- Résolution des erreurs SnapStart pour les fonctions Lambda

Quand utiliser SnapStart

Lambda SnapStart est conçu pour remédier à la variabilité de latence introduite par un code d'initialisation unique, tel que le chargement de dépendances de modules ou de frameworks. Ces opérations peuvent parfois prendre plusieurs secondes lors de l’invocation initiale. SnapStart À utiliser pour réduire cette latence de quelques secondes à une valeur inférieure à une seconde, dans des scénarios optimaux. SnapStart fonctionne mieux lorsqu'il est utilisé avec des invocations de fonctions à grande échelle. Les fonctions qui sont rarement invoquées peuvent ne pas bénéficier des mêmes améliorations de performance.

SnapStart est particulièrement utile pour deux principaux types d'applications :

- Flux utilisateur APIs et sensibles à la latence : les fonctions faisant partie des points de terminaison critiques des API ou des flux destinés aux utilisateurs peuvent bénéficier de la réduction de la latence et de SnapStart l'amélioration des temps de réponse.
- Flux de traitement des données sensibles à la latence : les flux de traitement des données limités dans le temps qui utilisent des fonctions Lambda peuvent améliorer le débit en réduisant le temps de latence d’initialisation des fonctions anormales.

La simultanéité allouée permet aux fonctions d’être initialisées et prêtes à réagir en quelques millisecondes. Utilisez la simultanéité provisionnée si votre application a des exigences strictes en matière de latence de démarrage à froid auxquelles il est impossible de répondre de manière adéquate. SnapStart

Fonctions prises en charge et limitations

SnapStart est disponible pour les environnements d'exécution gérés par Lambda suivants :

- Java 11 et versions ultérieures
- Python 3.12 et versions ultérieures
- .NET 8 et versions ultérieures. Si vous utilisez le framework Lambda Annotations pour .NET, passez à Amazon.Lambda.Annotations version 1.6.0 ou ultérieure pour garantir la compatibilité avec. SnapStart

Les autres environnements d'exécution gérés (tels que `nodejs24.x` et `ruby3.4`), Exécutions uniquement basées sur le système d'exploitationet les images de conteneurs ne sont pas pris en charge.

SnapStart ne prend pas en charge la simultanéité provisionnée, Amazon Elastic File System (Amazon EFS) ou le stockage éphémère supérieur à 512 Mo.

> **Note**
>
> Vous ne pouvez l'utiliser SnapStart que sur des versions de fonctions publiées et des alias pointant vers des versions. Vous ne pouvez pas l'utiliser SnapStart sur la version non publiée d'une fonction ($LATEST).

Régions prises en charge

Lambda SnapStart est disponible dans toutes les régions commerciales, à l'exception de l'Asie-Pacifique (Nouvelle Zélande) et de l'Asie-Pacifique (Taipei).

Considérations de compatibilité

Avec SnapStart, Lambda utilise un seul instantané comme état initial pour plusieurs environnements d'exécution. Si votre fonction utilise l'un des éléments suivants pendant la phase d'initialisation, vous devrez peut-être apporter quelques modifications avant de l'utiliser SnapStart :

Unicité

Si votre code d'initialisation génère un contenu unique qui est inclus dans l'instantané, ce contenu peut ne pas être unique lorsqu'il est réutilisé dans plusieurs environnements d'exécution. Pour conserver l'unicité lors de l'utilisation SnapStart, vous devez générer un contenu unique après l'initialisation. Cela inclut des secrets uniques et uniques IDs, ainsi que l'entropie utilisée pour

générer du pseudo-hasard. Pour savoir comment restaurer l'unicité, consultez Gestion de l'unicité avec Lambda SnapStart.

Connexions réseau

L'état des connexions que votre fonction établit pendant la phase d'initialisation n'est pas garanti lorsque Lambda reprend votre fonction à partir d'un instantané. Validez l'état de vos connexions réseau et rétablissez-les si nécessaire. Dans la plupart des cas, les connexions réseau établies par un AWS SDK reprennent automatiquement. Pour les autres connexions, consultez les bonnes pratiques.

Données temporaires

Certaines fonctions téléchargent ou initialisent des données éphémères, telles que des informations d'identification temporaires ou des horodatages mis en cache, pendant la phase d'initialisation. Actualisez les données éphémères dans le gestionnaire de fonctions avant de les utiliser, même lorsque vous ne les utilisez pas. SnapStart

SnapStart tarification

> **Note**
>
> Pour les environnements d'exécution gérés par Java, il n'y a aucun coût supplémentaire pour SnapStart. Vous êtes facturé en fonction du nombre de demandes pour vos fonctions, de la durée d'exécution de votre code et de la mémoire configurée pour votre fonction.

Le coût d'utilisation SnapStart inclut les éléments suivants :

- Mise en cache : pour chaque version de fonction que vous publiez avec cette SnapStart option activée, vous payez les frais de mise en cache et de maintenance de l'instantané. Le prix dépend de la quantité de mémoire que vous allouez à votre fonction. Vous êtes facturé pour un minimum de 3 heures. Vous continuerez à payer des frais tant que votre fonction restera active. Utilisez l'action ListVersionsByFunctionAPI pour identifier les versions des fonctions, puis utilisez-la DeleteFunctionpour supprimer les versions inutilisées. Pour supprimer automatiquement les versions de fonction inutilisées, consultez le modèle Lambda Version Cleanup sur Serverless Land.
- Restauration : chaque fois qu'une instance de fonction est restaurée à partir d'un instantané, vous payez des frais de restauration. Le prix dépend de la quantité de mémoire que vous allouez à votre fonction.

Comme pour toutes les fonctions Lambda, des frais de durée s'appliquent au code exécuté dans le gestionnaire de fonctions. Pour les SnapStart fonctions, les frais de durée s'appliquent également au code d'initialisation déclaré en dehors du gestionnaire, au temps nécessaire au chargement de l'exécution et à tout code exécuté dans un hook d'exécution. La durée est calculée à partir du moment où votre code commence à s'exécuter jusqu'à son retour ou sa fin, arrondie à la milliseconde la plus proche. Lambda conserve des copies en cache de votre instantané pour des raisons de résilience et leur applique automatiquement les mises à jour logicielles, telles que les mises à niveau de l'environnement d'exécution et les correctifs de sécurité. Des frais s'appliquent chaque fois que Lambda réexécute votre code d'initialisation pour appliquer des mises à jour logicielles.

Pour plus d'informations sur le coût d'utilisation SnapStart, consultez la section AWS Lambda Tarification.

Activation et gestion de Lambda SnapStart

Pour l'utiliser SnapStart, activez-la SnapStart sur une fonction Lambda nouvelle ou existante. Ensuite, publiez et invoquez une version de la fonction.

Rubriques

- Activation SnapStart (console)
- Activation SnapStart (AWS CLI)
- Activation SnapStart (API)
- Lambda SnapStart et états des fonctions
- Mise à jour d’un instantané
- Utilisation SnapStart avec AWS SDKs
- Utilisation SnapStart avec CloudFormationAWS SAM, et AWS CDK
- Suppression d’instantanés

Activation SnapStart (console)

SnapStart Pour activer une fonction

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom d’une fonction.
3. Choisissez Configuration, puis General configuration (Configuration générale).
4. Dans le volet General configuration (Configuration générale), choisissez Edit (Modifier).
5. Sur la page Modifier les paramètres de base, pour SnapStart, choisissez Versions publiées.
6. Choisissez Enregistrer.
7. Publiez une version de la fonction. Lambda initialise votre code, crée un instantané de l’environnement d’exécution initialisé, puis met en cache l’instantané pour un accès à faible latence.
8. Invoquez la version de la fonction..

Activation SnapStart (AWS CLI)

SnapStart Pour activer une fonction existante

1. Mettez à jour la configuration de la fonction en exécutant la update-function-configurationcommande avec l'--snap-startoption.

```
aws lambda update-function-configuration \
  --function-name my-function \
  --snap-start ApplyOn=PublishedVersions
```

2. Publiez une version de la fonction avec la commande publish-version.

```
aws lambda publish-version \
  --function-name my-function
```

3. Vérifiez qu'elle SnapStart est activée pour la version de la fonction en exécutant la get-function-configurationcommande et en spécifiant le numéro de version. L'exemple suivant spécifie la version 1.

```
aws lambda get-function-configuration \
  --function-name my-function:1
```

   Si la réponse indique que OptimizationStatusc'est le cas On et que l'état l' SnapStart estActive, alors il est activé et un instantané est disponible pour la version de fonction spécifiée.

```
"SnapStart": {
    "ApplyOn": "PublishedVersions",
    "OptimizationStatus": "On"
 },
 "State": "Active",
```

4. Invoquez la version de la fonction en exécutant la commande invoke et en spécifiant la version. L'exemple suivant invoque la version 1.

```
aws lambda invoke \
  --cli-binary-format raw-in-base64-out \
  --function-name my-function:1 \
  --payload '{ "name": "Bob" }' \
  response.json
```

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

À activer SnapStart lorsque vous créez une nouvelle fonction

1. Créez une fonction en exécutant la commande create-function avec l'option --snap-start. Pour --role, spécifiez le Amazon Resource Name (ARN) de votre rôle d'exécution.

```
aws lambda create-function \
  --function-name my-function \
  --runtime "java25" \
  --zip-file fileb://my-function.zip \
  --handler my-function.handler \
  --role arn:aws:iam::111122223333:role/lambda-ex \
  --snap-start ApplyOn=PublishedVersions
```

2. Créez une version avec la commande publish-version.

```
aws lambda publish-version \
  --function-name my-function
```

3. Vérifiez qu'elle SnapStart est activée pour la version de la fonction en exécutant la get-function-configurationcommande et en spécifiant le numéro de version. L'exemple suivant spécifie la version 1.

```
aws lambda get-function-configuration \
  --function-name my-function:1
```

Si la réponse indique que OptimizationStatusc'est le cas On et que l'état l' SnapStart est`Active`, alors il est activé et un instantané est disponible pour la version de fonction spécifiée.

```
"SnapStart": {
     "ApplyOn": "PublishedVersions",
     "OptimizationStatus": "On"
  },
```

```
  "State": "Active",
```

4. Invoquez la version de la fonction en exécutant la commande invoke et en spécifiant la version. L'exemple suivant invoque la version 1.

```
aws lambda invoke \
  --cli-binary-format raw-in-base64-out \
  --function-name my-function:1 \
  --payload '{ "name": "Bob" }' \
  response.json
```

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

Activation SnapStart (API)

Pour activer SnapStart

1. Effectuez l'une des actions suivantes :
   - Créez une nouvelle fonction SnapStart activée en utilisant l'action CreateFunctionAPI avec le SnapStartparamètre.
   - SnapStart Activez une fonction existante en utilisant l'UpdateFunctionConfigurationaction associée au SnapStartparamètre.
2. Publiez une version de fonction avec l'PublishVersionaction. Lambda initialise votre code, crée un instantané de l'environnement d'exécution initialisé, puis met en cache l'instantané pour un accès à faible latence.
3. Vérifiez qu'elle SnapStart est activée pour la version de la fonction en utilisant l'GetFunctionConfigurationaction. Spécifiez un numéro de version pour confirmer qu' SnapStart il est activé pour cette version. Si la réponse indique que OptimizationStatusc'est le cas 0n et que l'état l' SnapStart est`Active`, alors il est activé et un instantané est disponible pour la version de fonction spécifiée.

```
"SnapStart": {
        "ApplyOn": "PublishedVersions",
```

```
        "OptimizationStatus": "On"
    },
    "State": "Active",
```

4. Invoquez la version de la fonction avec l'action Invoke.

Lambda SnapStart et états des fonctions

Les états de fonction suivants peuvent se produire lorsque vous utilisez SnapStart.

En attente

Lambda initialise votre code et prend un instantané de l'environnement d'exécution initialisé. Les invocations ou autres actions d'API qui agissent sur la version de fonction échoueront.

Actif

La création de l'instantané est terminée et vous pouvez invoquer la fonction. Pour l'utiliser SnapStart, vous devez invoquer la version publiée de la fonction, et non la version non publiée ($LATEST).

Inactif

L'état `Inactive` peut se produire lorsque Lambda régénère périodiquement des instantanés de fonction pour appliquer des mises à jour logicielles. Dans ce cas, si votre fonction ne parvient pas à s'initialiser, elle peut entrer dans un état `Inactive`.

Pour les fonctions utilisant un environnement d'exécution Java, Lambda supprime les instantanés au bout de 14 jours sans invocation. Si vous invoquez la version de la fonction après 14 jours, Lambda renvoie une réponse `SnapStartNotReadyException` et commence à initialiser un nouvel instantané. Attendez que la fonction version atteigne l'état `Active`, puis invoquez-la à nouveau.

Échec

Lambda a rencontré une erreur lors de l'exécution du code d'initialisation ou de la création de l'instantané.

Mise à jour d'un instantané

Lambda crée un instantané pour chaque version de fonction publiée. Pour mettre à jour un instantané, publiez une nouvelle version de la fonction.

Utilisation SnapStart avec AWS SDKs

Pour effectuer des appels au AWS SDK depuis votre fonction, Lambda génère un ensemble éphémère d'informations d'identification en assumant le rôle d'exécution de votre fonction. Ces informations d'identification sont disponibles en tant que variables d'environnement lors de l'invocation de votre fonction. Vous n'avez pas besoin de fournir des informations d'identification pour le kit SDK directement dans le code. Par défaut, la chaîne de fournisseurs d'informations d'identification vérifie séquentiellement chaque endroit où vous pouvez définir des informations d'identification et sélectionne le premier disponible, généralement les variables d'environnement (`AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY` et `AWS_SESSION_TOKEN`).

> **Note**
>
> Lorsqu'il SnapStart est activé, le moteur d'exécution Lambda utilise automatiquement les informations d'identification du conteneur (`AWS_CONTAINER_CREDENTIALS_FULL_URI`et`AWS_CONTAINER_AUTHORIZATION_TOKEN`) au lieu des variables d'environnement de la clé d'accès. Cela évite que les informations d'identification n'expirent avant la restauration de la fonction.

Utilisation SnapStart avec CloudFormationAWS SAM, et AWS CDK

- AWS CloudFormation: Déclarez l'SnapStartentité dans votre modèle.
- AWS Serverless Application Model (AWS SAM) : Déclarez la SnapStartpropriété dans votre modèle.
- AWS Cloud Development Kit (AWS CDK): Utilisez le SnapStartPropertytype.

Suppression d'instantanés

Lambda supprime les instantanés lorsque :

- Vous supprimez la fonction ou la version de la fonction.
- Exécutions Java uniquement : vous n'invoquez pas la version de la fonction pendant 14 jours. Après 14 jours sans invocation, la version de fonction passe à l'état Inactif. Si vous invoquez la version de la fonction après 14 jours, Lambda renvoie une réponse `SnapStartNotReadyException` et commence à initialiser un nouvel instantané. Attendez que la version de la fonction atteigne l'état Actif, puis invoquez-la à nouveau.

Lambda supprime toutes les ressources associées aux instantanés supprimés, conformément au Règlement général sur la protection des données (RGPD).

Gestion de l'unicité avec Lambda SnapStart

Lorsque les invocations augmentent sur une fonction SnapStart, Lambda utilise un seul instantané initialisé pour reprendre plusieurs environnements d'exécution. Si votre code d'initialisation génère un contenu unique qui est inclus dans l'instantané, ce contenu peut ne pas être unique lorsqu'il est réutilisé dans plusieurs environnements d'exécution. Pour maintenir l'unicité lorsque vous utilisez SnapStart, vous devez générer un contenu unique après l'initialisation. Cela inclut les identifiants uniques, les secrets uniques et l'entropie utilisée pour générer un pseudo aléatoire.

Nous vous recommandons les bonnes pratiques suivantes pour vous aider à maintenir l'unicité de votre code. Pour les fonctions Java, Lambda fournit également un outil d'analyse SnapStart open source pour aider à vérifier le code qui suppose l'unicité. Si vous générez des données uniques pendant la phase d'initialisation, vous pouvez utiliser un hook d'exécution pour rétablir l'unicité. Avec les hooks d'exécution, vous pouvez exécuter un code spécifique immédiatement avant que Lambda ne prenne un instantané ou immédiatement après que Lambda ait repris une fonction à partir d'un instantané.

Évitez de sauvegarder un état qui dépend de l'unicité pendant l'initialisation

Pendant la phase d'initialisation de votre fonction, évitez de mettre en cache des données qui sont censées être uniques, comme la génération d'un ID unique pour la journalisation ou la configuration de seeds pour les fonctions random. Nous vous recommandons plutôt de générer des données uniques ou de configurer des seeds pour des fonctions random dans votre gestionnaire de fonction ou d'utiliser un hook d'exécution.

Les exemples suivants montrent comment générer un UUID dans le gestionnaire de fonctions.

Java

Example – Génération d'un ID unique dans le gestionnaire de fonctions

```
import java.util.UUID;
  public class Handler implements RequestHandler<String, String> {
    private static UUID uniqueSandboxId = null;
    @Override
    public String handleRequest(String event, Context context) {
      if (uniqueSandboxId == null)
        uniqueSandboxId = UUID.randomUUID();
      System.out.println("Unique Sandbox Id: " + uniqueSandboxId);
      return "Hello, World!";
```

```
    }
  }
```

Python

Example – Génération d'un ID unique dans le gestionnaire de fonctions

```
import json
import random
import time

unique_number = None

def lambda_handler(event, context):
    seed = int(time.time() * 1000)
    random.seed(seed)
    global unique_number
    if not unique_number:
        unique_number = random.randint(1, 10000)

    print("Unique number: ", unique_number)

    return "Hello, World!"
```

.NET

Example – Génération d'un ID unique dans le gestionnaire de fonctions

```
namespace Example;
public class SnapstartExample
{
    private Guid _myExecutionEnvironmentGuid;
    public SnapstartExample()
    {
        // This GUID is set for non-restore use cases, such as testing or if
 SnapStart is turned off
        _myExecutionEnvironmentGuid = new Guid();
        // Register the method which will run after each restore. You may need to
 update Amazon.Lambda.Core to see this
        Amazon.Lambda.Core.SnapshotRestore.RegisterAfterRestore(MyAfterRestore);
    }

    private ValueTask MyAfterRestore()
    {
```

```
        // After restoring this snapshot to a new execution environment, update the
 GUID
        _myExecutionEnvironmentGuid = new Guid();
        return ValueTask.CompletedTask;
    }

    public string Handler()
    {
        return $"Hello World! My Execution Environment GUID is
 {_myExecutionEnvironmentGuid}";
    }
}
```

Utilisation de générateurs de nombres pseudo aléatoires cryptographiquement sécurisés (CSPRNG)

Si votre application dépend du caractère aléatoire, nous vous recommandons d'utiliser des générateurs de nombres aléatoires cryptographiquement sécurisés (CSPRNG). Outre OpenSSL 1.0.2, les environnements d'exécution gérés par Lambda incluent également les CSPRNG intégrés suivants :

- Java : `java.security.SecureRandom`
- Python: `random.SystemRandom`
- .NET : `System.Security.Cryptography.RandomNumberGenerator`

Les logiciels qui obtiennent toujours des nombres aléatoires de `/dev/random` ou `/dev/urandom` maintiennent également le caractère aléatoire avec SnapStart.

Les bibliothèques de chiffrement AWS conservent automatiquement le caractère aléatoire avec SnapStart en commençant par les versions minimales spécifiées dans le tableau suivant. Si vous utilisez ces bibliothèques avec vos fonctions Lambda, assurez-vous d'utiliser les versions minimales ou ultérieures suivantes :

Bibliothèque	Version minimale prise en charge (x86)	Version minimale prise en charge (ARM)
AWS libcrypto (AWS-LC)	1.16.0	1.30.0

Bibliothèque	Version minimale prise en charge (x86)	Version minimale prise en charge (ARM)
AWS libcrypto FIPS	2.0.13	2.0.13

Si vous empaquetez les bibliothèques de chiffrement ci-dessus avec vos fonctions Lambda sous forme de dépendances transitives via les bibliothèques suivantes, assurez-vous d'utiliser les versions minimales suivantes ou des versions ultérieures :

Bibliothèque	Version minimale prise en charge (x86)	Version minimale prise en charge (ARM)
AWS SDK for Java 2.x	2.23.20	2.26.12
AWS Common Runtime pour Java	0.29.8	0.29.25
Amazon Corretto Crypto Provider	2.4.1	2.4.1
Amazon Corretto Crypto Provider FIPS	2.4.1	2.4.1

Les exemples suivants montrent comment utiliser de CSPRNG pour garantir des séquences de nombres uniques même lorsque la fonction est restaurée à partir d'un instantané.

Java

Example – java.security.SecureRandom

```
import java.security.SecureRandom;
  public class Handler implements RequestHandler<String, String> {
    private static SecureRandom rng = new SecureRandom();
    @Override
    public String handleRequest(String event, Context context) {
      for (int i = 0; i < 10; i++) {
        System.out.println(rng.next());
      }
      return "Hello, World!";
```

```
    }
  }
```

Python

Example – random.SystemRandom

```
import json
import random

secure_rng = random.SystemRandom()

def lambda_handler(event, context):
    random_numbers = [secure_rng.random() for _ in range(10)]

    for number in random_numbers:
        print(number)

    return "Hello, World!"
```

.NET

Example – RandomNumberGenerator

```
using Amazon.Lambda.Core;
using System.Security.Cryptography;
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeriali

namespace DotnetSecureRandom;

public class Function
{
    public string FunctionHandler()
    {
        using (RandomNumberGenerator rng = RandomNumberGenerator.Create())
        {
            byte[] randomUnsignedInteger32Bytes = new byte[4];
            for (int i = 0; i < 10; i++)
            {
                rng.GetBytes(randomUnsignedInteger32Bytes);
                int randomInt32 = BitConverter.ToInt32(randomUnsignedInteger32Bytes,
 0);
```

```
                Console.WriteLine("{0:G}", randomInt32);
            }
        }
        return "Hello World!";
    }
}
```

Outil d'analyse de SnapStart (Java uniquement)

Lambda fournit un outil d'analyse pour Java pour vous aider à vérifier le code qui suppose l'unicité. L'outil d'analyse SnapStart est un plugin SpotBugs open source qui exécute une analyse statique par rapport à un ensemble de règles. L'outil d'analyse aide à identifier les implémentations de code potentielles qui pourraient enfreindre les suppositions concernant l'unicité. Pour obtenir des instructions d'installation et une liste des vérifications effectuées par l'outil d'analyse, consultez le référentiel aws-lambda-snapstart-java-rules sur GitHub.

Pour en savoir plus sur la gestion de l'unicité avec SnapStart, consultez Starting up faster with AWS Lambda SnapStart sur le blog AWS Compute.

Implémentation du code avant ou après les instantanés de fonctions Lambda

Vous pouvez utiliser des hooks d'exécution pour implémenter du code avant que Lambda ne crée un instantané ou après que Lambda ait repris une fonction à partir d'un instantané. Les hooks d'exécution sont utiles à diverses fins, telles que :

- Le nettoyage et l'initialisation : avant de créer un instantané, vous pouvez utiliser un hook d'exécution pour effectuer des opérations de nettoyage ou de libération de ressources. Après la restauration d'un instantané, vous pouvez utiliser un hook d'exécution pour réinitialiser les ressources ou les états qui n'ont pas été capturés dans l'instantané.
- La configuration dynamique : vous pouvez utiliser des hooks d'exécution pour mettre à jour dynamiquement la configuration ou d'autres métadonnées avant la création d'un instantané ou après sa restauration. Cela peut être utile si votre fonction doit s'adapter aux modifications de l'environnement d'exécution.
- Les intégrations externes : vous pouvez utiliser des hooks d'exécution pour intégrer des services ou des systèmes externes, tels que l'envoi de notifications ou la mise à jour de l'état externe, dans le cadre du processus de point de contrôle et de restauration.
- L'optimisation des performances : vous pouvez utiliser des hooks d'exécution pour optimiser la séquence de démarrage de votre fonction, par exemple en préchargeant les dépendances. Pour de plus amples informations, consultez Personnalisation de performances.

Les pages suivantes expliquent comment implémenter des hooks d'exécution pour votre environnement d'exécution préféré.

Rubriques

- Hooks d'exécution Lambda SnapStart pour Java
- Hooks d'exécution Lambda SnapStart pour Python
- Hooks d'exécution Lambda SnapStart pour .NET

Hooks d'exécution Lambda SnapStart pour Java

Vous pouvez utiliser des hooks d'exécution pour implémenter du code avant que Lambda ne crée un instantané ou après que Lambda ait repris une fonction à partir d'un instantané. Les hooks d'exécution sont disponibles dans le cadre du projet open source Coordinated Restore at Checkpoint

(CRaC). CRaC est en cours de développement pour l'Open Java Development Kit (OpenJDK). Pour un exemple d'utilisation de CRaC avec une application de référence, consultez le référentiel CRaC sur GitHub. CRaC utilise trois éléments principaux :

- `Resource` – Une interface avec deux méthodes, `beforeCheckpoint()` et `afterRestore()`. Utilisez ces méthodes pour implémenter le code que vous voulez exécuter avant un instantané et après une restauration.
- `Context <R extends Resource>` – Pour recevoir des notifications pour les points de contrôle et les restaurations, une `Resource` doit être enregistrée avec un `Context`.
- `Core` – Le service de coordination, qui fournit le `Context` global par défaut via la méthode statique `Core.getGlobalContext()`.

Pour plus d'informations sur `Context` et `Resource`, consultez Package org.crac dans la documentation de CRaC.

Suivez les étapes suivantes pour implémenter des hooks d'exécution avec le package org.crac. L'environnement d'exécution Lambda contient une implémentation de contexte CRaC personnalisée qui appelle vos hooks d'exécution avant le point de contrôle et après la restauration.

Enregistrement et exécution du hook d'exécution

L'ordre dans lequel Lambda exécute vos hooks d'exécution est déterminé par l'ordre d'enregistrement. L'ordre d'enregistrement suit l'ordre d'importation, de définition ou d'exécution dans votre code.

- `beforeCheckpoint()` : exécutés dans l'ordre inverse de l'enregistrement
- `afterRestore()` : exécutés dans l'ordre d'enregistrement

Assurez-vous que tous les hooks enregistrés sont correctement importés et inclus dans le code de votre fonction. Si vous enregistrez les hooks d'exécution dans un fichier ou un module distinct, vous devez vous assurer que le module est importé, soit directement, soit dans le cadre d'un package plus large, dans le fichier de gestionnaire de votre fonction. Si le fichier ou le module n'est pas importé dans le gestionnaire de fonctions, Lambda ignore les hooks d'exécution.

> **Note**
>
> Lorsque Lambda crée un instantané, votre code d'initialisation peut s'exécuter jusqu'à 15 minutes. Le délai d'attente est de 130 secondes ou le délai d'expiration de la fonction

configurée (900 secondes au maximum), la valeur la plus élevée étant retenue. Vos hooks d'exécution `beforeCheckpoint()` sont comptabilisés dans le délai d'attente du code d'initialisation. Lorsque Lambda restaure un instantané, l'exécution doit se charger et les hooks d'exécution `afterRestore()` doivent se terminer dans le délai imparti (10 secondes). Sinon, vous obtiendrez une SnapStartTimeoutException.

Étape 1 : mettre à jour de configuration de création

Ajoutez la dépendance `org.crac` à la configuration de compilation. L'exemple suivant utilise Gradle. Pour des exemples concernant d'autres systèmes de compilation, consultez la documentation Apache Maven.

```
dependencies {
    compile group: 'com.amazonaws', name: 'aws-lambda-java-core', version: '1.2.1'
    # All other project dependecies go here:
    # ...
    # Then, add the org.crac dependency:
 implementation group: 'org.crac', name: 'crac', version: '1.4.0'
}
```

Étape 2 : mettre à jour le gestionnaire Lambda

Le gestionnaire de fonction Lambda est la méthode dans votre code de fonction qui traite les événements. Lorsque votre fonction est invoquée, Lambda exécute la méthode du gestionnaire. Votre fonction s'exécute jusqu'à ce que le gestionnaire renvoie une réponse, se ferme ou expire.

Pour de plus amples informations, consultez Définition du gestionnaire de fonction Lambda dans Java.

L'exemple de gestionnaire suivant montre comment exécuter le code avant le point de contrôle (`beforeCheckpoint()`) et après la restauration (`afterRestore()`). Ce gestionnaire enregistre également la `Resource` dans le `Context` global géré par l'environnement d'exécution.

> **Note**
>
> Lorsque Lambda crée un instantané, votre code d'initialisation peut s'exécuter jusqu'à 15 minutes. Le délai d'attente est de 130 secondes ou le délai d'expiration de la fonction configurée (900 secondes au maximum), la valeur la plus élevée étant retenue. Vos hooks

d'exécution `beforeCheckpoint()` sont comptabilisés dans le délai d'attente du code d'initialisation. Lorsque Lambda restaure un instantané, l'exécution (JVM) doit se charger et les hooks d'exécution `afterRestore()` doivent se terminer dans le délai imparti (10 secondes). Sinon, vous obtiendrez une SnapStartTimeoutException.

```
...
  import org.crac.Resource;
  import org.crac.Core;
  ...
public class CRaCDemo implements RequestStreamHandler, Resource {
    public CRaCDemo() {
      Core.getGlobalContext().register(this);
    }
    public String handleRequest(String name, Context context) throws IOException {
      System.out.println("Handler execution");
      return "Hello " + name;
    }
    @Override
    public void beforeCheckpoint(org.crac.Context<? extends Resource> context)
        throws Exception {
      System.out.println("Before checkpoint");
    }
    @Override
    public void afterRestore(org.crac.Context<? extends Resource> context)
        throws Exception {
      System.out.println("After restore");
```

`Context` maintient uniquement une `WeakReference` à l'objet enregistré. Si une `Resource` est récupérée par le récupérateur de mémoire, les hooks d'exécution ne s'exécutent pas. Votre code doit maintenir une référence forte à la `Resource` pour garantir l'exécution du hook d'exécution.

Voici deux exemples de modèles à éviter :

Example – Objet sans référence forte

```
Core.getGlobalContext().register( new MyResource() );
```

Example – Objets de classes anonymes

```
Core.getGlobalContext().register( new Resource() {
```

```
    @Override
    public void afterRestore(Context<? extends Resource> context) throws Exception {
     // ...
    }

    @Override
    public void beforeCheckpoint(Context<? extends Resource> context) throws Exception {
     // ...
    }

} );
```

Au lieu de cela, maintenez une référence forte. Dans l'exemple suivant, la ressource enregistrée n'est pas récupérée par le récupérateur de mémoire et les hooks d'exécution s'exécutent de manière cohérente.

Example – Objet avec une référence forte

```
Resource myResource = new MyResource(); // This reference must be maintained to prevent
 the registered resource from being garbage collected
Core.getGlobalContext().register( myResource );
```

Hooks d'exécution Lambda SnapStart pour Python

Vous pouvez utiliser des hooks d'exécution pour implémenter du code avant que Lambda ne crée un instantané ou après que Lambda ait repris une fonction à partir d'un instantané. Les hooks d'exécution Python sont disponibles dans le cadre de la bibliothèque Snapshot Restore for Python open source, qui est incluse dans les environnements d'exécution gérés par Python. Cette bibliothèque fournit deux décorateurs que vous pouvez utiliser pour définir vos hooks d'exécution :

- `@register_before_snapshot` : pour les fonctions que vous souhaitez exécuter avant que Lambda ne crée un instantané.
- `@register_after_restore` : pour les fonctions que vous souhaitez exécuter lorsque Lambda reprend une fonction à partir d'un instantané.

Vous pouvez également utiliser les méthodes suivantes afin d'enregistrer des fonctions pouvant être appelées pour des hooks d'exécution :

- `register_before_snapshot(func, *args, **kwargs)`

- `register_after_restore(func, *args, **kwargs)`

Enregistrement et exécution du hook d'exécution

L'ordre dans lequel Lambda exécute vos hooks d'exécution est déterminé par l'ordre d'enregistrement :

- Avant l'instantané : exécutés dans l'ordre inverse de l'enregistrement
- Après l'instantané : exécutés dans l'ordre d'enregistrement

L'ordre d'enregistrement des hooks d'exécution dépend de la façon dont vous définissez les hooks. Lorsque vous utilisez des décorateurs (`@register_before_snapshot` et `@register_after_restore`), l'ordre d'enregistrement suit l'ordre d'importation, de définition ou d'exécution dans votre code. Si vous avez besoin de plus de contrôle sur l'ordre d'enregistrement, utilisez les méthodes `register_before_snapshot()` et `register_after_restore()` au lieu des décorateurs.

Assurez-vous que tous les hooks enregistrés sont correctement importés et inclus dans le code de votre fonction. Si vous enregistrez les hooks d'exécution dans un fichier ou un module distinct, vous devez vous assurer que le module est importé, soit directement, soit dans le cadre d'un package plus large, dans le fichier de gestionnaire de votre fonction. Si le fichier ou le module n'est pas importé dans le gestionnaire de fonctions, Lambda ignore les hooks d'exécution.

> **Note**
>
> Lorsque Lambda crée un instantané, votre code d'initialisation peut s'exécuter jusqu'à 15 minutes. Le délai d'attente est de 130 secondes ou le délai d'expiration de la fonction configurée (900 secondes au maximum), la valeur la plus élevée étant retenue. Vos hooks d'exécution `@register_before_snapshot` sont comptabilisés dans le délai d'attente du code d'initialisation. Lorsque Lambda restaure un instantané, l'exécution doit se charger et les hooks d'exécution `@register_after_restore` doivent se terminer dans le délai imparti (10 secondes). Sinon, vous obtiendrez une SnapStartTimeoutException.

Exemple

L'exemple de gestionnaire suivant montre comment exécuter le code avant le point de contrôle (`@register_before_snapshot`) et après la restauration (`@register_after_restore`).

```
from snapshot_restore_py import register_before_snapshot, register_after_restore

def lambda_handler(event, context):
    # Handler code

@register_before_snapshot
def before_checkpoint():
    # Logic to be executed before taking snapshots

@register_after_restore
def after_restore():
    # Logic to be executed after restore
```

Pour plus d'exemples, consultez Snapshot Restore pour Python dans le référentiel GitHub AWS.

Hooks d'exécution Lambda SnapStart pour .NET

Vous pouvez utiliser des hooks d'exécution pour implémenter du code avant que Lambda ne crée un instantané ou après que Lambda ait repris une fonction à partir d'un instantané. Les hooks d'exécution .NET sont disponibles dans le cadre du package Amazon.Lambda.Core (version 2.5.0 ou ultérieure). Cette bibliothèque fournit deux méthodes que vous pouvez utiliser pour définir vos hooks d'exécution :

- `RegisterBeforeSnapshot()` : code à exécuter avant la création de l'instantané
- `RegisterAfterSnapshot()` : code à exécuter après la reprise d'une fonction à partir d'un instantané

> **Note**
>
> Si vous utilisez le cadre Lambda Annotations pour .NET, passez à Amazon.Lambda.Annotations version 1.6.0 ou ultérieure pour garantir la compatibilité avec SnapStart.

Enregistrement et exécution du hook d'exécution

Enregistrez vos hooks dans votre code d'initialisation. Tenez compte des directives suivantes en fonction du modèle d'exécution de votre fonction Lambda :

- Pour l'approche de l'assemblage exécutable, enregistrez vos hooks avant de démarrer l'amorçage Lambda avec `RunAsync`.
- Pour l'approche de la bibliothèque de classes, enregistrez vos hooks dans le constructeur de classe du gestionnaire.
- Pour les applications ASP.NET Core, enregistrez vos hooks avant d'appeler la méthode `WebApplications.Run`.

Pour enregistrer des hooks d'exécution pour SnapStart dans .NET, utilisez les méthodes suivantes :

```
Amazon.Lambda.Core.SnapshotRestore.RegisterBeforeSnapshot(BeforeCheckpoint);
Amazon.Lambda.Core.SnapshotRestore.RegisterAfterRestore(AfterCheckpoint);
```

Lorsque plusieurs types de hooks sont enregistrés, l'ordre dans lequel Lambda exécute vos hooks d'exécution est déterminé par l'ordre d'enregistrement :

- `RegisterBeforeSnapshot()` : exécutés dans l'ordre inverse de l'enregistrement
- `RegisterAfterSnapshot()` : exécutés dans l'ordre d'enregistrement

> **Note**
>
> Lorsque Lambda crée un instantané, votre code d'initialisation peut s'exécuter jusqu'à 15 minutes. Le délai d'attente est de 130 secondes ou le délai d'expiration de la fonction configurée (900 secondes au maximum), la valeur la plus élevée étant retenue. Vos hooks d'exécution `RegisterBeforeSnapshot()` sont comptabilisés dans le délai d'attente du code d'initialisation. Lorsque Lambda restaure un instantané, l'exécution doit se charger et les hooks d'exécution `RegisterAfterSnapshot()` doivent se terminer dans le délai imparti (10 secondes). Sinon, vous obtiendrez une SnapStartTimeoutException.

Exemple

L'exemple de fonction suivant montre comment exécuter le code avant le point de contrôle (`RegisterBeforeSnapshot`) et après la restauration (`RegisterAfterRestore`).

```
public class SampleClass
{
    public SampleClass()
```

```
    {
        Amazon.Lambda.Core.SnapshotRestore.RegisterBeforeSnapshot(BeforeCheckpoint);
        Amazon.Lambda.Core.SnapshotRestore.RegisterAfterRestore(AfterCheckpoint);
    }

    private ValueTask BeforeCheckpoint()
    {
        // Add logic to be executed before taking the snapshot
        return ValueTask.CompletedTask;
    }

    private ValueTask AfterCheckpoint()
    {
        // Add logic to be executed after restoring the snapshot
        return ValueTask.CompletedTask;
    }

    public APIGatewayProxyResponse FunctionHandler(APIGatewayProxyRequest request,
 ILambdaContext context)
    {
        // Add business logic

        return new APIGatewayProxyResponse
        {
            StatusCode = 200
        };
    }
}
```

Surveillance pour Lambda SnapStart

Vous pouvez surveiller vos fonctions Lambda SnapStart en utilisant Amazon CloudWatch, AWS X-Ray et l'API Accès aux données de télémétrie en temps réel pour les extensions à l'aide de l'API de télémétrie.

> **Note**
>
> Les variables d'environnement `AWS_LAMBDA_LOG_GROUP_NAME` et `AWS_LAMBDA_LOG_STREAM_NAME` ne sont pas disponibles dans les fonctions SnapStart Lambda.

Comprendre le comportement de journalisation et de facturation avec SnapStart

Il existe quelques différences avec le format du flux de journal CloudWatch pour les fonctions SnapStart :

- Journaux d'initialisation – Lorsqu'un nouvel environnement d'exécution est créé, le `REPORT` n'inclut pas le champ `Init Duration`. C'est parce que Lambda initialise les fonctions SnapStart lorsque vous créez une version au lieu de le faire pendant l'invocation de la fonction. Pour les fonctions SnapStart, le champ `Init Duration` se trouve dans l'enregistrement `INIT_REPORT`. Cet enregistrement indique les détails de la durée du Phase d'initialisation, y compris la durée de tout hook d'exécution `beforeCheckpoint`.
- Journaux d'invocation – Lorsqu'un nouvel environnement d'exécution est créé, le `REPORT` inclut les champs `Restore Duration` et `Billed Restore Duration` :
  - `Restore Duration` : le temps nécessaire à Lambda pour restaurer un instantané, charger l'exécution et exécuter les éventuels hooks d'exécution après restauration. Le processus de restauration des instantanés peut inclure du temps consacré à des activités en dehors de la MicroVM. Ce temps est indiqué dans `Restore Duration`.
  - `Billed Restore Duration` : le temps nécessaire à Lambda pour charger l'exécution et exécuter les éventuels hooks d'exécution après restauration.

> **Note**
>
> Comme pour toutes les fonctions Lambda, des frais de durée s'appliquent au code exécuté dans le gestionnaire de fonctions. Pour les fonctions SnapStart, les frais de durée s'appliquent également au code d'initialisation déclaré en dehors du gestionnaire, au temps de chargement de l'environnement d'exécution et à tout code s'exécutant dans un hook d'exécution.

La durée du démarrage à froid est la somme de `Restore Duration` + `Duration`.

L'exemple suivant est une demande Lambda Insights qui renvoie les percentiles de latence pour les fonctions SnapStart. Pour plus d'informations sur les demandes Lambda Insights, consultez Exemple de flux de travail utilisant des demandes pour dépanner une fonction.

```
filter @type = "REPORT"
  | parse @log /\d+:\/aws\/lambda\/(?<function>.*)/
  | parse @message /Restore Duration: (?<restoreDuration>.*?) ms/
  | stats
count(*) as invocations,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 50) as p50,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 90) as p90,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 99) as p99,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 99.9) as p99.9
group by function, (ispresent(@initDuration) or ispresent(restoreDuration)) as
 coldstart
  | sort by coldstart desc
```

Suivi actif de X-Ray pour SnapStart

Vous pouvez utiliser X-Ray pour suivre les demandes adressées aux fonctions Lambda SnapStart. Il existe quelques différences avec les sous-segments X-Ray pour les fonctions SnapStart :

- Il n'existe aucun sous-segment `Initialization` pour les fonctions SnapStart.
- Le sous-segment `Restore` indique le temps nécessaire à Lambda pour restaurer un instantané, charger l'exécution et exécuter les éventuels hooks d'exécution après restauration. Le processus de restauration des instantanés peut inclure du temps consacré à des activités en dehors de la MicroVM. Cette heure est indiquée dans le sous-segment `Restore`. Le temps passé en dehors de la microVM pour restaurer un instantané ne vous est pas facturé.

Événements de l'API de télémétrie pour SnapStart

Lambda envoie les événements SnapStart suivants à API de télémétrie :

- `platform.restoreStart` – Indique l'heure à laquelle la phase `Restore` a commencé.
- `platform.restoreRuntimeDone` – Indique si la phase `Restore` a réussi. Lambda envoie ce message lorsque l'environnement d'exécution envoie une demande d'API d'exécution `restore/next`. Il existe trois statuts possibles : succès, échec et dépassement de délai.
- `platform.restoreReport` – Indique la durée de la phase `Restore` et le nombre de millisecondes pour lesquelles vous avez été facturé pendant cette phase.

Métriques de l'URL de la fonction et de l'Amazon API Gateway

Si vous créez une API Web à l'aide d'API Gateway, vous pouvez utiliser la métrique IntegrationLatency pour mesurer la latence de bout en bout (le temps entre le moment où API Gateway relaie une demande au backend et celui où il reçoit une réponse du backend).

Si vous utilisez une URL de la fonction Lambda, vous pouvez utiliser la métrique UrlRequestLatency pour mesurer la latence de bout en bout (le temps entre le moment où l'URL de la fonction reçoit une demande et celui où l'URL de la fonction renvoie une réponse).

Modèle de sécurité pour Lambda SnapStart

Lambda SnapStart prend en charge le chiffrement au repos. Lambda chiffre les instantanés avec un AWS KMS key. Par défaut, Lambda utilise une Clé gérée par AWS. Si ce comportement par défaut convient à votre flux, vous n'avez pas besoin de configurer autre chose. Sinon, vous pouvez utiliser l'option `--kms-key-arn` dans la commande create-function ou update-function-configuration pour fournir une clé AWS KMS gérée par le client. Vous pouvez procéder ainsi pour contrôler la rotation de la clé KMS ou pour répondre aux exigences de votre organisation en matière de gestion des clés KMS. Les clés gérés par le client entraînent des frais AWS KMS standard. Pour plus d'informations, consultez Tarification d'AWS Key Management Service.

Lorsque vous supprimez une fonction ou une version de fonction SnapStart, toutes les demandes `Invoke` à cette fonction ou version de fonction échouent. Lambda supprime toutes les ressources associées aux instantanés supprimés, conformément au Règlement général sur la protection des données (RGPD).

Optimisation des performances Lambda SnapStart

Rubriques

- Personnalisation de performances
- Bonnes pratiques concernant le réseau

Personnalisation de performances

Pour optimiser les avantages de SnapStart, tenez compte des recommandations d'optimisation de code suivantes pour votre environnement d'exécution.

> **Note**
>
> SnapStart fonctionne de façon optimale lorsqu'il est utilisé avec des invocations de fonctions à grande échelle. Les fonctions qui sont rarement invoquées peuvent ne pas bénéficier des mêmes améliorations de performance.

Java

Pour tirer le meilleur parti de SnapStart, nous vous recommandons de précharger les dépendances et d'initialiser les ressources qui contribuent à la latence de démarrage dans votre code d'initialisation plutôt que dans le gestionnaire de fonction. Cela permet de déplacer la latence associée au chargement lourd des classes hors du chemin d'appel, optimisant ainsi les performances de start-up avec SnapStart.

Si vous ne pouvez pas précharger les dépendances ou les ressources pendant l'initialisation, nous vous recommandons de les précharger avec des invocations fictives. Pour ce faire, mettez à jour le code du gestionnaire de fonction, comme le montre l'exemple suivant tiré de la fonction d'animalerie sur le référentiel GitHub de AWS Labs.

```
private static SpringLambdaContainerHandler<AwsProxyRequest, AwsProxyResponse> handler;
  static {
      try {
          handler =
 SpringLambdaContainerHandler.getAwsProxyHandler(PetStoreSpringAppConfig.class);

          // Use the onStartup method of the handler to register the custom filter
```

```
            handler.onStartup(servletContext -> {
                FilterRegistration.Dynamic registration =
servletContext.addFilter("CognitoIdentityFilter", CognitoIdentityFilter.class);
                registration.addMappingForUrlPatterns(EnumSet.of(DispatcherType.REQUEST),
false, "/*");
            });

            // Send a fake Amazon API Gateway request to the handler to load classes
ahead of time
            ApiGatewayRequestIdentity identity = new ApiGatewayRequestIdentity();
            identity.setApiKey("foo");
            identity.setAccountId("foo");
            identity.setAccessKey("foo");

            AwsProxyRequestContext reqCtx = new AwsProxyRequestContext();
            reqCtx.setPath("/pets");
            reqCtx.setStage("default");
            reqCtx.setAuthorizer(null);
            reqCtx.setIdentity(identity);

            AwsProxyRequest req = new AwsProxyRequest();
            req.setHttpMethod("GET");
            req.setPath("/pets");
            req.setBody("");
            req.setRequestContext(reqCtx);

            Context ctx = new TestContext();
            handler.proxy(req, ctx);


        } catch (ContainerInitializationException e) {
            // if we fail here. We re-throw the exception to force another cold start
            e.printStackTrace();
            throw new RuntimeException("Could not initialize Spring framework", e);
        }
    }
```

Python

Pour tirer le meilleur parti de SnapStart, concentrez-vous sur l'organisation efficace du code et la gestion des ressources au sein de vos fonctions Python. De manière générale, effectuez les tâches de calcul intensives pendant la phase d'initialisation. Cette approche élimine les opérations fastidieuses du processus d'invocation, améliorant ainsi les performances globales des fonctions.

Pour implémenter cette stratégie de manière efficace, nous vous recommandons les bonnes pratiques suivantes :

- Importez les dépendances en dehors du gestionnaire de fonction.
- Créez des instances `boto3` en dehors du gestionnaire.
- Initialisez les ressources ou les configurations statiques avant que le gestionnaire ne soit invoqué.
- Envisagez d'utiliser un hook d'exécution avant l'instantané pour les tâches gourmandes en ressources, comme le téléchargement de fichiers externes, le préchargement de frameworks tels que Django ou le chargement de modèles de machine learning.

Example — Optimisation de la fonction Python pour SnapStart

```
# Import all dependencies outside of Lambda handler
from snapshot_restore_py import register_before_snapshot
import boto3
import pandas
import pydantic

# Create S3 and SSM clients outside of Lambda handler
s3_client = boto3.client("s3")

# Register the function to be called before snapshot
@register_before_snapshot
def download_llm_models():
    # Download an object from S3 and save to tmp
    # This files will persist in this snapshot
    with open('/tmp/FILE_NAME', 'wb') as f:
        s3_client.download_fileobj('amzn-s3-demo-bucket', 'OBJECT_NAME', f)
    ...

def lambda_handler(event, context):
    ...
```

.NET

Pour réduire le temps de compilation juste-à-temps et de chargement de l'assemblage, pensez à invoquer votre gestionnaire de fonctions à partir d'un hook d'exécution `RegisterBeforeCheckpoint`. En raison du fonctionnement de la compilation par niveau .NET, vous obtiendrez des résultats optimaux en invoquant le gestionnaire plusieurs fois, comme dans l'exemple suivant.

> **Important**
>
> Assurez-vous que l'invocation de votre fonction fictive ne produit pas d'effets secondaires imprévus, tels que le lancement de transactions commerciales.

Example

```
public class Function
{
    public Function()
    {
        Amazon.Lambda.Core.SnapshotRestore.RegisterBeforeSnapshot(FunctionWarmup);
    }

    // Warmup method that calls the function handler before snapshot to warm up
 the .NET code and runtime.
    // This speeds up future cold starts after restoring from a snapshot.

    private async ValueTask FunctionWarmup()
    {
        var request = new APIGatewayProxyRequest
        {
            Path = "/heathcheck",
            HttpMethod = "GET"
        };

        for (var i = 0; i < 10; i++)
        {
            await FunctionHandler(request, null);
        }
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request, ILambdaContext context)
    {
        //
        // Process HTTP request
        //

        var response = new APIGatewayProxyResponse
        {
            StatusCode = 200
```

```
        };

        return await Task.FromResult(response);
    }
}
```

Bonnes pratiques concernant le réseau

L'état des connexions que votre fonction établit pendant la phase d'initialisation n'est pas garanti lorsque Lambda reprend votre fonction à partir d'un instantané. Dans la plupart des cas, les connexions réseau qu'un kit SDK AWS établit reprennent automatiquement. Pour les autres connexions, nous vous recommandons les bonnes pratiques suivantes.

Rétablissez les connexions réseau

Rétablissez toujours vos connexions réseau lorsque votre fonction reprend à partir d'un instantané. Nous vous recommandons de rétablir les connexions réseau dans le gestionnaire de fonction. Vous pouvez également utiliser un hook d'exécution après la restauration.

N'utilisez pas le nom d'hôte comme identifiant unique d'environnement d'exécution

Nous vous déconseillons d'utiliser `hostname` pour identifier votre environnement d'exécution comme un nœud ou un conteneur unique dans vos applications. Avec SnapStart, un seul instantané est utilisé comme état initial pour plusieurs environnements d'exécution. Tous les environnements d'exécution renvoient la même valeur `hostname` pour `InetAddress.getLocalHost()` (Java), `socket.gethostname()` (Python) et `Dns.GetHostName()` (.NET). Pour les applications qui nécessitent une identité ou une valeur `hostname` d'environnement d'exécution unique, nous vous recommandons de générer un identifiant unique dans le gestionnaire de fonction. Ou bien, utilisez un hook d'exécution après la restauration pour générer un ID unique, puis utilisez cet ID unique comme identifiant de l'environnement d'exécution.

Évitez de lier les connexions à des ports source fixes

Nous vous recommandons d'éviter de lier les connexions réseau à des ports source fixes. Les connexions sont rétablies lorsqu'une fonction reprend à partir d'un instantané, et les connexions réseau qui sont liées à un port source fixe peuvent échouer.

Évitez d'utiliser le cache DNS de Java

Les fonctions Lambda mettent déjà en cache les réponses DNS. Si vous utilisez un autre cache DNS avec SnapStart, vous risquez de subir des interruptions de connexion lorsque la fonction reprend à partir d'un instantané.

La classe `java.util.logging.Logger` peut activer indirectement le cache DNS de la JVM. Pour remplacer les paramètres par défaut, définissez networkaddress.cache.ttl sur 0 avant de procéder à l'initialisation de `logger`. Exemple :

```
public class MyHandler {
  // first set TTL property
  static{
   java.security.Security.setProperty("networkaddress.cache.ttl" , "0");
  }
 // then instantiate logger
  var logger = org.apache.logging.log4j.LogManager.getLogger(MyHandler.class);
}
```

Pour éviter les échecs `UnknownHostException` de l'environnement d'exécution de Java 11, nous vous recommandons de définir la valeur de `networkaddress.cache.negative.ttl` sur 0. Dans les environnements d'exécution de Java 17 et versions ultérieures, cette étape n'est pas nécessaire. Vous pouvez définir cette propriété pour une fonction Lambda à l'aide de la variable d'environnement `AWS_LAMBDA_JAVA_NETWORKADDRESS_CACHE_NEGATIVE_TTL=0`.

La désactivation du cache DNS de la JVM ne désactive pas la mise en cache DNS gérée par Lambda.

Résolution des erreurs SnapStart pour les fonctions Lambda

Cette page aborde les problèmes courants qui se produisent lors de l'utilisation de Lambda SnapStart, notamment les erreurs de création d'instantanés, les erreurs de délai d'expiration et les erreurs de service internes.

SnapStartNotReadyException

Erreur : une erreur s'est produite (SnapStartNotReadyException) lors de l'appel de l'opération Invoke20150331 : Lambda est en train d'initialiser votre fonction. L'invocation sera possible une fois que l'état de votre fonction deviendra ACTIVE.

Causes courantes

Cette erreur se produit lorsque vous essayez d'invoquer une version de fonction qui se trouve à l'état `Inactive`. La version de votre fonction devient `Inactive` lorsqu'elle n'a pas été invoquée depuis 14 jours ou lorsque Lambda recycle périodiquement l'environnement d'exécution.

Résolution

Attendez que la fonction version atteigne l'état `Active`, puis invoquez-la à nouveau.

SnapStartTimeoutException

Problème : vous recevez le message `SnapStartTimeoutException` lorsque vous essayez d'invoquer une version de fonction SnapStart.

Cause courante

Pendant la phase de restauration, Lambda restaure l'environnement d'exécution Java et exécute tous les hooks d'exécution après la restauration. Si un hook d'exécution après la restauration s'exécute pendant plus de 10 secondes, la phase `Restore` expire et vous obtenez une erreur lorsque vous essayez d'invoquer la fonction. Les problèmes de connexion réseau et d'identification peuvent également entraîner des délais d'attente de la phase `Restore`.

Résolution

Consultez les journaux CloudWatch de la fonction pour détecter les erreurs de délai d'expiration survenues pendant la phase de restauration. Assurez-vous que tous les hooks après restauration se terminent en moins de 10 secondes.

Example Journal CloudWatch

```
{ "cause": "Lambda couldn't restore the snapshot within the timeout limit. (Service:
 Lambda, Status Code: 408, Request ID: 11a222c3-410f-427c-ab22-931d6bcbf4f2)", "error":
 "Lambda.SnapStartTimeoutException"}
```

Erreur interne du service 500

Erreur : Lambda n'a pas pu créer de nouvel instantané car vous avez atteint votre limite de création d'instantanés simultanés.

Cause courante

Une erreur 500 est une erreur interne au sein du service Lambda lui-même, plutôt qu'un problème lié à votre fonction ou à votre code. Ces erreurs sont souvent intermittentes.

Résolution

Réessayez de publier la version de la fonction.

401 Accès non autorisé

Erreur : jeton de session ou clé d'en-tête incorrect

Cause courante

Cette erreur se produit lors de l'utilisation de l'AWS Systems Manager Parameter Store et de l'extension AWS Secrets Manager avec Lambda SnapStart.

Résolution

L'AWS Systems Manager Parameter Store et l'extension AWS Secrets Manager ne sont pas compatibles avec SnapStart. L'extension génère des informations d'identification pour communiquer avec AWS Secrets Manager lors de l'initialisation de la fonction, ce qui entraîne des erreurs d'expiration des informations d'identification lorsqu'elle est utilisée avec SnapStart.

UnknownHostException (Java)

Erreur : impossible d'exécuter la requête HTTP : le certificat pour `abc.us-east-1.amazonaws.com` ne correspond à aucun des noms alternatifs d'objet.

Cause courante

Les fonctions Lambda mettent déjà en cache les réponses DNS. Si vous utilisez un autre cache DNS avec SnapStart, vous risquez de subir des interruptions de connexion lorsque la fonction reprend à partir d'un instantané.

Résolution

Pour éviter les échecs `UnknownHostException` de l'environnement d'exécution de Java 11, nous vous recommandons de définir la valeur de `networkaddress.cache.negative.ttl` sur 0. Dans les environnements d'exécution de Java 17 et versions ultérieures, cette étape n'est pas nécessaire. Vous pouvez définir cette propriété pour une fonction Lambda à l'aide de la variable d'environnement `AWS_LAMBDA_JAVA_NETWORKADDRESS_CACHE_NEGATIVE_TTL=0`.

Échecs de création d'instantanés

Erreur : AWS Lambda n'a pas pu invoquer votre fonction SnapStart. Si cette erreur persiste, vérifiez la présence d'erreurs d'initialisation dans les journaux CloudWatch de votre fonction.

Résolution

Consultez les journaux Amazon CloudWatch Logs de votre fonction pour connaître les délais d'expiration du hook d'exécution avant le point de contrôle. Vous pouvez également essayer de publier une nouvelle version de fonction, ce qui peut parfois résoudre le problème.

Latence de création d'instantanés

Problème : lorsque vous publiez une nouvelle version de fonction, celle-ci reste à l'état `Pending` pendant une longue période.

Cause courante

Lorsque Lambda crée un instantané, votre code d'initialisation peut s'exécuter jusqu'à 15 minutes. Le délai d'attente est de 130 secondes ou le délai d'expiration de la fonction configurée (900 secondes au maximum), la valeur la plus élevée étant retenue.

Si votre fonction est attachée à un VPC, Lambda devra peut-être également créer des interfaces réseau avant que la fonction ne devienne `Active`. Si vous essayez d'invoquer la version de la fonction alors que la fonction est `Pending`, vous pourriez obtenir une erreur 409

`ResourceConflictException`. Si la fonction est invoquée à l'aide d'un point de terminaison Amazon API Gateway, une erreur 500 peut s'afficher dans API Gateway.

Résolution

Attendez au moins 15 minutes que la version de la fonction soit initialisée avant de l'invoquer.

Isolement des locataires

Utilisez le mode d'isolation des locataires lorsque vous avez besoin de traiter des demandes isolées pour des utilisateurs finaux individuels ou des locataires invoquant une fonction Lambda. Cette fonctionnalité simplifie la création d'applications mutualisées qui traitent du code ou des données spécifiques au locataire, telles que les plateformes SaaS pour l'automatisation des flux de travail ou l'exécution de code, en supprimant la nécessité de gérer les ressources fonctionnelles spécifiques au locataire et la logique de routage des demandes.

Les applications multi-locataires sont soumises à des exigences d'isolation strictes lorsqu'elles exécutent du code ou traitent des données pour des locataires individuels ou des utilisateurs finaux. Avec le mode d'isolation des locataires, Lambda utilise un identifiant de locataire spécifié par le client pour acheminer les demandes vers les environnements d'exécution des fonctions sous-jacents, garantissant ainsi que les environnements d'exécution d'une fonction ne sont utilisés que pour traiter les appels provenant de l'utilisateur final ou du locataire spécifié. Les environnements d'exécution des fonctions de Lambda tirent parti de la virtualisation Firecracker pour isoler les charges de travail.

Lorsqu'une fonction utilisant le mode d'isolation du locataire reçoit un appel avec un identifiant de locataire, Lambda tente d'abord de localiser un environnement d'exécution disponible associé à cet identifiant de locataire. S'il n'existe aucun environnement d'exécution, Lambda crée et attribue un nouvel environnement d'exécution à ce locataire. À mesure que les invocations de fonctions avec l'identifiant de locataire spécifié augmentent, Lambda localise ou crée de nouveaux environnements d'exécution selon les besoins.

Rubriques

- Quand utiliser le mode d'isolation des locataires
- Fonctions prises en charge et limitations
- Soutenu Régions AWS
- Considérations
- Tarification
- Mode d'isolation
- Activation de l'isolation des locataires pour les fonctions Lambda
- Invocation de fonctions Lambda avec isolation des locataires
- Accès à l'identifiant du locataire dans le code de fonction Lambda
- Surveillance des fonctions Lambda avec isolation des locataires

- Résolution des problèmes liés à l'isolation des locataires pour les fonctions Lambda

Quand utiliser le mode d'isolation des locataires

Utilisez le mode d'isolation des locataires lorsque vous devez servir plusieurs utilisateurs finaux ou locataires à l'aide d'une seule fonction Lambda, tout en veillant à ce que les environnements d'exécution utilisés pour traiter les appels destinés à des locataires individuels restent isolés les uns des autres. Cette isolation stricte des environnements d'exécution est requise pour les applications multi-locataires qui :

- Exécuter le code fourni par l'utilisateur final : le maintien d'environnements d'exécution distincts pour les clients individuels peut limiter l'impact de l'exécution de code fourni par l'utilisateur qui peut être incorrect ou malveillant.
- Traitement des données spécifiques au locataire : le maintien d'environnements d'exécution distincts pour chaque locataire peut empêcher l'exposition de données sensibles spécifiques au locataire à d'autres locataires.

Plusieurs demandes d'invocation émanant du même locataire peuvent réutiliser le même environnement d'exécution de fonctions, réduisant ainsi les démarrages à froid et améliorant les temps de réponse pour les applications sensibles à la latence.

Fonctions prises en charge et limitations

Le mode d'isolation du locataire n'est pas pris en charge avec les fonctions qui utilisent la fonction URLs, la simultanéité provisionnée ou. SnapStart Vous pouvez envoyer des demandes à une fonction isolée par le locataire à l'aide d'invocations synchrones, d'invocationsasynchrones ou en utilisant Amazon API Gateway comme déclencheur d'événement.

Soutenu Régions AWS

Le mode d'isolation des locataires est pris en charge dans toutes les régions commerciales, à l'exception de l'Asie-Pacifique (Nouvelle Zélande).

Considérations

Lorsque vous utilisez l'isolation des locataires avec vos fonctions Lambda, gardez à l'esprit les points suivants :

- Configuration immuable : l'isolation des locataires est une propriété de fonction immuable. Elle ne peut être activée que lors de la création d'une fonction.
- Paramètre tenant-id requis : les fonctions utilisant le mode d'isolation des locataires doivent être invoquées avec un `tenant-id` paramètre. L'omission de ce paramètre entraînera l'échec des appels de fonctions.
- Le rôle d'exécution s'applique à tous les locataires : les invocations provenant de tous les locataires utilisent les autorisations définies dans le rôle d'exécution de votre fonction Lambda.
- Concurrence : aucun changement n'est apporté au comportement simultané ou à la mise à l'échelle de votre fonction lorsque vous utilisez l'isolation des locataires. Lambda impose une limite de 2 500 environnements d'exécution isolés par les locataires (actifs ou inactifs) pour 1 000 exécutions simultanées configurées pour votre fonction Lambda.

Tarification

Vous êtes débité lorsque Lambda crée un nouvel environnement d'exécution isolé par les locataires. Le prix dépend de la quantité de mémoire que vous allouez à votre fonction et de l'architecture du processeur que vous utilisez. Pour en savoir plus, consultez Pricing AWS Lambda (Tarification).

Mode d'isolation

Le tableau suivant décrit les différences entre les fonctions Lambda avec et sans isolation des locataires.

Fonctionnalité	Avec isolation des locataires	Sans isolement des locataires
Type d'isolation	Isolation au niveau du locataire	Isolation au niveau des fonctions
Réutilisation de l'environnement	Les environnements d'exécution ne sont jamais réutilisés entre différents locataires	Les environnements d'exécution peuvent être réutilisés lors des invocations de la même fonction

Fonctionnalité	Avec isolation des locataires	Sans isolement des locataires
Isolation des données	Les données des autres locataires ne sont pas accessibles	Les données des appels précédents de la même version de fonction peuvent être accessibles
Démarrages à froid	Davantage de démarrages à froid en raison des environnements spécifiques aux locataires	Moins de démarrages à froid grâce à la réutilisation dans l'environnement
Tarification	Frais supplémentaires en plus de la tarification Lambda standard	Tarification Lambda standard

Activation de l'isolation des locataires pour les fonctions Lambda

Pour activer le mode d'isolation des locataires, créez une nouvelle fonction Lambda. Vous ne pouvez pas activer l'isolation des locataires sur les fonctions existantes.

Rubriques

- Activation de l'isolation des locataires (console)
- Permettre l'isolement des locataires (AWS CLI)
- Activation de l'isolation des locataires (API)
- Permettre l'isolement des locataires (CloudFormation)

Activation de l'isolation des locataires (console)

Pour créer une fonction Lambda à l'aide de la console

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez Créer une fonction.
3. Sélectionnez Créer à partir de zéro.
4. Dans le volet Informations de base, pour Nom de la fonction, saisissez **`image-analysis`**.
5. Pour Runtime, choisissez l'un des environnements d'exécution Lambda pris en charge.
6. Sous Configurations supplémentaires, mode d'isolation des locataires, sélectionnez Activer.
7. Vérifiez vos paramètres, puis choisissez Créer une fonction.

Permettre l'isolement des locataires (AWS CLI)

Créez une fonction en isolant les locataires

Lorsque vous créez une nouvelle fonction à l'aide de la CLI, ajoutez l'`--tenancy-config '{"TenantIsolationMode": "PER_TENANT"}'` option à votre demande de création de fonction. Exemple :

```
aws lambda create-function \
    --function-name image-analysis \
    --runtime nodejs24.x \
    --zip-file fileb://image-analysis-function.zip \
    --handler image-analysis-function.handler \
```

```
    --role arn:aws:iam:123456789012:role/execution-role \
    --tenancy-config '{"TenantIsolationMode": "PER_TENANT"}'
```

Activation de l'isolation des locataires (API)

Pour activer l'isolation des locataires à l'aide de l'API Lambda

1. Créez une nouvelle fonction avec l'isolation des locataires activée en utilisant l'action CreateFunctionAPI avec le `TenancyConfig` paramètre.
2. Vérifiez que l'isolation des locataires est activée pour la fonction en utilisant l'GetFunctionConfigurationaction. Si la réponse indique que `TenantIsolationMode` c'est le cas`PER_TENANT`, l'isolation des locataires est activée pour la fonction :

   ```
   "TenancyConfig": {
           "TenantIsolationMode": "PER_TENANT"
       }
   ```

Invoquez la version de la fonction avec l'action Invoke. Pour de plus amples informations, veuillez consulter Invocation de fonctions Lambda avec isolation des locataires.

Permettre l'isolement des locataires (CloudFormation)

Le CloudFormation modèle suivant crée une nouvelle fonction Lambda avec l'isolation des locataires activée :

```
MyLambdaFunction:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: my-sample-python-lambda
      Runtime: python3.14
      Role: !GetAtt LambdaExecutionRole.Arn
      Handler: index.lambda_handler
      TenancyConfig:
        TenantIsolationMode: PER_TENANT
      Code:
        ZipFile: |
          import json

          def lambda_handler(event, context):
              return {
```

```
                'statusCode': 200,
                'body': json.dumps(f'Hello from Lambda! Tenant-ID:
{context.tenant_id}')
            }
     Timeout: 10
     MemorySize: 128
```

Invocation de fonctions Lambda avec isolation des locataires

Lorsque vous appelez une fonction pour laquelle l'isolation des locataires est activée, vous devez fournir un `tenant-id` paramètre. Ce paramètre garantit que l'appel de votre fonction est traité dans un environnement d'exécution dédié à ce locataire spécifique.

Invocation de fonctions avec isolation des locataires ()AWS CLI

Invocation synchrone

Pour les appels synchrones, ajoutez le `--tenant-id` paramètre à votre commande Invoke :

```
aws lambda invoke \
    --function-name image-analysis \
    --tenant-id blue \
    response.json
```

invocation asynchrone

Pour les appels asynchrones, incluez à la fois les paramètres et : `--tenant-id --invocation-type Event`

```
aws lambda invoke \
    --function-name image-analysis \
    --tenant-id blue \
    --invocation-type Event \
    response.json
```

Invocation de fonctions avec isolation des locataires (API)

Lorsque vous utilisez directement l'action d'API Invoke, incluez l'identifiant du locataire à l'aide du `X-Amzn-Tenant-Id` paramètre dans votre demande.

Exemple de demande d'API

```
POST /2015-03-31/functions/image-analysis/invocations HTTP/1.1
Host: lambda.us-east-1.amazonaws.com
Content-Type: application/json
Authorization: AWS4-HMAC-SHA256 Credential=...
X-Amz-Tenant-Id: blue
```

```
{
    "key1": "value1",
    "key2": "value2"
}
```

Invocation de fonctions avec isolation des locataires (API Gateway)

Lorsque vous utilisez API Gateway REST APIs pour déclencher des fonctions Lambda isolées par les locataires, vous devez configurer API Gateway pour mapper les propriétés des demandes du client à l'en-tête `X-Amz-Tenant-Id` attendu par Lambda. API Gateway utilise l'action d'API Invoke de Lambda, qui nécessite que l'ID du locataire soit transmis à l'aide de l'en-tête `X-Amz-Tenant-Id` HTTP. Vous pouvez configurer API Gateway pour injecter cet en-tête HTTP dans la demande d'appel Lambda avec une valeur obtenue à partir des propriétés de demande du client telles que les en-têtes HTTP, les paramètres de requête ou les paramètres de chemin. Vous devez d'abord mapper la propriété de la demande du client avant de pouvoir remplacer l'`X-Amz-Tenant-Id`en-tête.

> **Note**
>
> Vous ne pouvez pas utiliser le protocole HTTP APIs pour appeler des fonctions Lambda isolées par les locataires, car il n'est pas possible de remplacer l'en-tête. `X-Amz-Tenant-Id`

Utilisation des en-têtes de demande

Configurez votre intégration API Gateway pour mapper un en-tête personnalisé entre la demande du client et l'`X-Amz-Tenant-Id`en-tête. L'exemple suivant montre une demande client avec un `x-tenant-id` en-tête :

```
POST /api/process HTTP/1.1
Host: your-api-id.execute-api.us-east-1.amazonaws.com
Content-Type: application/json
x-tenant-id: blue

{
    "data": "sample payload"
}
```

Dans la configuration de votre méthode API Gateway, vous devez :

1. Activez le paramètre d'en-tête de demande du client (par exemple,`method.request.header.x-tenant-id`)
2. Mappez l'en-tête du client à l'en-tête d'intégration Lambda en utilisant `integration.request.header.X-Amz-Tenant-Id`

Utilisation des paramètres de requête

De même, vous pouvez associer les paramètres de requête à l'`X-Amz-Tenant-Id`en-tête :

```
GET /api/process?tenant-id=blue&data=sample HTTP/1.1
Host: your-api-id.execute-api.us-east-1.amazonaws.com
```

Configurez la méthode pour activer le paramètre de requête et mappez-le à l'en-tête d'intégration.

Invocation de fonctions avec isolation des locataires (SDK)

Lorsque vous l'utilisez AWS SDKs pour invoquer des fonctions isolées par le locataire, incluez l'identifiant du locataire dans votre demande d'invocation.

Python

```
import boto3
import json

lambda_client = boto3.client('lambda')

response = lambda_client.invoke(
    FunctionName='image-analysis',
    TenantId='blue',
    Payload=json.dumps({
        'key1': 'value1',
        'key2': 'value2'
    })
)

result = json.loads(response['Payload'].read())
```

Node.js

```
const AWS = require('aws-sdk');
```

```
const lambda = new AWS.Lambda();

const params = {
    FunctionName: 'image-analysis',
    TenantId: 'blue',
    Payload: JSON.stringify({
        key1: 'value1',
        key2: 'value2'
    })
};

lambda.invoke(params, (err, data) => {
    if (err) {
        console.error(err);
    } else {
        const result = JSON.parse(data.Payload);
        console.log(result);
    }
});
```

Java

```
import software.amazon.awssdk.services.lambda.LambdaClient;
import software.amazon.awssdk.services.lambda.model.InvokeRequest;
import software.amazon.awssdk.services.lambda.model.InvokeResponse;
import software.amazon.awssdk.core.SdkBytes;

public class TenantIsolationExample {

    public static void main(String[] args) {
        LambdaClient lambdaClient = LambdaClient.create();

        String payload = "{\"key1\": \"value1\", \"key2\": \"value2\"}";

        InvokeRequest request = InvokeRequest.builder()
                .functionName("image-analysis")
                .tenantId("blue")
                .payload(SdkBytes.fromUtf8String(payload))
                .build();

        InvokeResponse response = lambdaClient.invoke(request);
    }
}
```

Accès à l'identifiant du locataire dans le code de fonction Lambda

Lorsque l'isolation des locataires est activée dans votre fonction Lambda, l'identifiant de locataire utilisé pour appeler votre fonction est mis à disposition dans l'objet de contexte transmis à votre gestionnaire de fonctions. Vous pouvez utiliser cet identifiant pour implémenter des fonctionnalités de logique, de surveillance et de débogage spécifiques au locataire.

Rubriques

- Accès à l'identifiant du locataire
- Modèles d'utilisation courants
- Surveillance et débogage

Accès à l'identifiant du locataire

L'identifiant du locataire est disponible via la `tenantId` propriété de l'objet de contexte. Notez que cette propriété est disponible pendant la phase d'appel, et non pendant la phase d'initialisation.

Python

```
def lambda_handler(event, context):
    tenant_id = context.tenant_id
    print(f"Processing request for tenant: {tenant_id}")

    # Implement tenant-specific logic
    if tenant_id == "blue":
        return process_blue_tenant(event)
    elif tenant_id == "green":
        return process_green_tenant(event)
    else:
        return process_default_tenant(event)
```

Node.js

```
exports.handler = async (event, context) => {
    const tenantId = context.tenantId;
    console.log(`Processing request for tenant: ${tenantId}`);

    // Implement tenant-specific logic
    switch (tenantId) {
        case 'blue':
```

```
            return processBlueTenant(event);
        case 'green':
            return processGreenTenant(event);
        default:
            return processDefaultTenant(event);
    }
};
```

Java

```
public class TenantHandler implements RequestHandler<Map<String, Object>, String> {

    @Override
    public String handleRequest(Map<String, Object> event, Context context) {
        String tenantId = context.getTenantId();
        System.out.println("Processing request for tenant: " + tenantId);

        // Implement tenant-specific logic
        switch (tenantId) {
            case "blue":
                return processBlueTenant(event);
            case "green":
                return processGreenTenant(event);
            default:
                return processDefaultTenant(event);
        }
    }
}
```

Modèles d'utilisation courants

Voici les méthodes les plus courantes pour utiliser l'identifiant du locataire dans votre code de fonction :

Configuration spécifique au locataire

Utilisez l'ID du locataire pour charger la configuration ou les paramètres spécifiques au locataire :

```
def lambda_handler(event, context):
    tenant_id = context.tenant_id

    # Load tenant-specific configuration
```

```
    config = load_tenant_config(tenant_id)
    database_url = config['database_url']
    api_key = config['api_key']

    # Process with tenant-specific settings
    return process_request(event, database_url, api_key)
```

Accès aux données spécifique au locataire

Utilisez l'ID du locataire pour garantir l'isolation des données et le contrôle d'accès :

```
import boto3

def lambda_handler(event, context):
    tenant_id = context.tenant_id

    # Ensure data access is scoped to the tenant
    dynamodb = boto3.resource('dynamodb')
    table = dynamodb.Table('user_data')

    user_id = event.get('userId')

    response = table.get_item(
        Key={
            'tenant_id': tenant_id,
            'user_id': user_id
        }
    )

    return process_results(response.get('Item'), tenant_id)
```

Surveillance et débogage

L'identifiant du locataire est automatiquement inclus dans les journaux Lambda lorsque la journalisation JSON est activée, ce qui facilite le suivi et le débogage des problèmes spécifiques au locataire. Vous pouvez également utiliser l'ID du locataire pour des mesures et un suivi personnalisés.

Example Métriques personnalisées avec identifiant de locataire

L'exemple suivant montre comment utiliser l'ID du locataire pour créer des CloudWatch métriques spécifiques au locataire afin de surveiller les modèles d'utilisation et les performances par locataire :

```
import boto3

def lambda_handler(event, context):
    tenant_id = context.tenant_id
    cloudwatch = boto3.client('cloudwatch')

    # Record tenant-specific metrics
    cloudwatch.put_metric_data(
        Namespace='MyApp/TenantMetrics',
        MetricData=[
            {
                'MetricName': 'RequestCount',
                'Dimensions': [
                    {
                        'Name': 'TenantId',
                        'Value': tenant_id
                    }
                ],
                'Value': 1,
                'Unit': 'Count'
            }
        ]
    )

    return process_request(event, tenant_id)
```

Surveillance des fonctions Lambda avec isolation des locataires

Vous pouvez surveiller vos fonctions Lambda isolées par les locataires à l'aide d'Amazon CloudWatch et en accédant aux données de télémétrie en temps réel pour les extensions à l'aide de l'API de télémétrie. AWS X-Ray

Comprendre la journalisation en mode isolé pour les locataires

Pour les fonctions utilisant l'isolation des locataires, Lambda inclut automatiquement l'identifiant du locataire dans les journaux des fonctions lorsque la journalisation JSON est activée, ce qui facilite le suivi et le débogage des problèmes spécifiques aux locataires. Lambda crée un flux de CloudWatch journal distinct pour chaque environnement d'exécution. Vous pouvez utiliser CloudWatch Logs Insights pour rechercher les flux de journaux appartenant à un locataire en particulier en filtrant par identifiant de locataire :

```
fields @logStream, @message
| filter tenantId=='BlueTenant' or record.tenantId=='BlueTenant'
| stats count() as logCount by @logStream
| sort @timestamp desc
```

Vous pouvez également utiliser ce paramètre pour récupérer les journaux spécifiques au locataire dans tous les flux de journaux :

```
fields @message
| filter tenantId=='BlueTenant' or record.tenantId=='BlueTenant'
| limit 1000
```

La `tenantId` propriété est incluse pour les événements des plateformes (tels que `platform.start` et`platform.report`) et les journaux personnalisés que vous imprimez dans votre code de fonction, comme indiqué ci-dessous :

```
{
    "time": "2025-10-13T19:48:06.990Z",
    "type": "platform.start",
    "record": {
        "requestId": "a0f40320-b43c-44b3-91bf-d5b5240a1bed",
        "functionArn": "arn:aws:lambda:us-east-1:xxxxxx:function:multitenant-
function-1",
        "version": "$LATEST",
```

```
        "tenantId": "BlueTenant""
    }
}
{
    "timestamp": "2025-10-13T19:48:06.992Z",
    "level": "INFO",
    "requestId": "a0f40320-b43c-44b3-91bf-d5b5240a1bed",
    "tenantId": "BlueTenant"",
    "message": "custom log line1"
}
{
    "timestamp": "2025-10-13T19:48:07.022Z",
    "level": "WARN",
    "requestId": "a0f40320-b43c-44b3-91bf-d5b5240a1bed",
    "tenantId": "BlueTenant"",
    "message": "custom log line2"
}
```

Résolution des problèmes liés à l'isolation des locataires pour les fonctions Lambda

Cette page aborde les problèmes courants qui se produisent lors de l'utilisation de l'isolation des locataires pour AWS Lambda.

InvalidParameterValueException

Erreur : la configuration de l'ID du locataire n'est pas spécifiée ou n'est pas transmise à une fonction lorsque l'isolation du locataire n'est pas activée.

Causes courantes

Cette erreur se produit lors de l'appel d'une fonction isolée par le locataire sans identifiant de locataire, ou lors de l'appel d'une non-tenant-isolated fonction avec un identifiant de locataire.

Résolution

Ajoutez un ID de locataire si l'isolation des locataires est activée dans la fonction, ou supprimez l'ID de locataire si l'isolation des locataires n'est pas activée dans la fonction.

TooManyRequestsException

Erreur : taux dépassé

Causes courantes

Outre la limitation du débit basée sur le nombre maximal d'exécutions simultanées et le taux de dimensionnement des fonctions, Lambda limite le nombre maximum d'environnements d'exécution sensibles aux locataires (actifs ou inactifs) pouvant exister à la fois à 2 500 pour 1 000 exécutions simultanées de votre fonction.

Résolution

Pour résoudre ce problème, vous pouvez soit réduire la fréquence à laquelle les demandes d'invocation avec des identifiants de locataire uniques sont effectuées, soit implémenter des nouvelles tentatives avec backoff et jitter, soit demander une augmentation de la limite de simultanéité des fonctions.

Invoquer Lambda avec des événements provenant d'autres services AWS

Certains Services AWS peuvent appeler directement des fonctions Lambda à l'aide de déclencheurs. Ces services envoient des événements à Lambda, et la fonction est invoquée aussitôt que l'événement spécifié se produit. Les déclencheurs sont adaptés aux événements discrets et au traitement en temps réel. Lorsque vous créez un déclencheur à l'aide de la console Lambda, celle-ci interagit avec le AWS service correspondant pour configurer la notification d'événement sur ce service. Le déclencheur est en fait stocké et géré par le service qui génère les événements, et non par Lambda.

Les événements sont des données structurées au format JSON. La structure JSON varie en fonction du service qui la génère et du type d'événement, mais elles contiennent toutes les données nécessaires à la fonction pour traiter l'événement.

Une fonction peut avoir plusieurs déclencheurs. Chaque déclencheur agit comme un client qui invoque votre fonction indépendamment, et chaque événement que Lambda transmet à votre fonction contient des données provenant d'un seul déclencheur. Lambda convertit le document d'événement en objet et le transmet à votre gestionnaire de fonctions.

Selon le service, l'invocation pilotée par l'événement peut être synchrone ou asynchrone.

- Pour une invocation synchrone, l'autre service qui génère l'événement attend la réponse de votre fonction. Ce service définit les données que la fonction doit renvoyer dans la réponse. Le service contrôle la stratégie d'erreur, par exemple s'il faut réessayer les erreurs.
- Pour l'invocation asynchrone, Lambda place l'événement dans une file d'attente avant de le transmettre à votre fonction. Lorsque Lambda met l'événement en file d'attent, il envoie immédiatement une réponse de réussite au service qui a généré l'événement. Après que la fonction ait traité l'événement, Lambda ne renvoie pas de réponse au service qui a généré l'événement.

Création d'un déclencheur

La manière la plus simple de créer un déclencheur consiste à utiliser la console Lambda. Lorsque vous créez un déclencheur à l'aide de la console, Lambda dernier ajoute automatiquement les autorisations requises à la politique basée sur les ressources de la fonction.

Pour créer un déclencheur à l'aide de la console Lambda

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction pour laquelle vous souhaitez créer un déclencheur.
3. Dans le volet de Présentation de la fonction, choisissez Ajouter un déclencheur.
4. Sélectionnez le AWS service pour lequel vous souhaitez appeler votre fonction.
5. Renseignez les options dans le volet de configuration du déclencheur et choisissez Ajouter. En fonction de la fonction Service AWS que vous choisissez d'invoquer, les options de configuration du déclencheur seront différentes.

Services pouvant appeler des fonctions Lambda

Le tableau suivant répertorie les services qui peuvent invoquer des fonctions Lambda.

Service	Méthode d'invocation
Streaming géré par Amazon pour Apache Kafka	Mappage des sources d'événements
Self-managed Apache Kafka	Mappage des sources d'événements
Amazon API Gateway	Pilotée par les événements ; invocation synchrone
AWS CloudFormation	Pilotée par les événements ; invocation asynchrone
Amazon CloudWatch Logs	Pilotée par les événements ; invocation asynchrone
AWS CodeCommit	Pilotée par les événements ; invocation asynchrone
AWS CodePipeline	Pilotée par les événements ; invocation asynchrone
Amazon Cognito	Pilotée par les événements ; invocation synchrone
AWS Config	Pilotée par les événements ; invocation asynchrone
Amazon Connect	Pilotée par les événements ; invocation synchrone
Amazon DocumentDB	Mappage des sources d'événements

Service	Méthode d'invocation
Amazon DynamoDB	Mappage des sources d'événements
Elastic Load Balancing (Application Load Balancer)	Pilotée par les événements ; invocation synchrone
Amazon EventBridge (CloudWatch Événements)	Pilotée par les événements ; invocation asynchrone (bus d'événements), invocation synchrone ou asynchrone (canaux et plannings)
AWS IoT	Pilotée par les événements ; invocation asynchrone
Amazon Kinesis	Mappage des sources d'événements
Amazon Data Firehose	Pilotée par les événements ; invocation synchrone
Amazon Lex	Pilotée par les événements ; invocation synchrone
Amazon MQ	Mappage des sources d'événements
Amazon Simple Email Service	Pilotée par les événements ; invocation asynchrone
Amazon Simple Notification Service	Pilotée par les événements ; invocation asynchrone
Amazon Simple Queue Service	Mappage des sources d'événements
Amazon Simple Storage Service (Amazon S3)	Pilotée par les événements ; invocation asynchrone
Amazon Simple Storage Service Batch	Pilotée par les événements ; invocation synchrone
Secrets Manager	Rotation des secrets
AWS Step Functions	Pilotée par les événements ; invocation synchrone ou asynchrone
Amazon VPC Lattice	Pilotée par les événements ; invocation synchrone

Utilisation de Lambda avec Apache Kafka

Lambda prend en charge Apache Kafka en tant que source d'événement. Apache Kafka est une plateforme open source de diffusion en continu d'événements conçue pour gérer des pipelines de données en temps réel à haut débit et des applications de diffusion en continu. Il existe deux méthodes principales pour utiliser Lambda avec Apache Kafka :

- the section called "MSK"— Amazon Managed Streaming pour Apache Kafka (Amazon MSK) est un service entièrement géré par. AWS Amazon MSK permet d'automatiser la gestion de votre infrastructure Kafka, notamment le provisionnement, l'application de correctifs et la mise à l'échelle.
- the section called "Self-managed Apache Kafka"— En AWS termes terminologiques, un cluster autogéré inclut les clusters Kafka non AWS hébergés. Par exemple, vous pouvez toujours utiliser Lambda avec un cluster Kafka hébergé par un fournisseur non cloud tel que Confluent AWS Cloud ou Redpanda.

Lorsque vous choisissez entre Amazon MSK et Apache Kafka autogéré, tenez compte de vos besoins opérationnels et de vos exigences en matière de contrôle. Amazon MSK est un meilleur choix si vous souhaitez vous aider AWS à gérer rapidement une configuration Kafka évolutive et prête pour la production avec un minimum de frais opérationnels. Cette solution simplifie la sécurité, la surveillance et la haute disponibilité, vous permettant ainsi de vous concentrer sur le développement d'applications plutôt que sur la gestion de l'infrastructure. D'autre part, Apache Kafka autogéré est mieux adapté aux cas d'utilisation exécutés sur des environnements non AWS hébergés, y compris les clusters sur site.

Rubriques

- Utilisation de Lambda avec Amazon MSK
- Utilisation de Lambda avec Apache Kafka autogéré
- Modes de mise à l'échelle des sondeurs d'événements Apache Kafka dans Lambda
- Positions de départ des interrogations et flux Apache Kafka dans Lambda
- ID de groupe de consommateurs personnalisable dans Lambda
- Filtrage des événements provenant d'Amazon MSK et sources d'événements Apache Kafka autogéré
- Utilisation de registres de schémas avec les sources d'événements Kafka dans Lambda
- Traitement à faible latence pour les sources d'événements Kafka
- Configuration des contrôles de gestion des erreurs pour les sources d'événements Kafka

- Capture de lots supprimés pour des sources d’événement Amazon MSK et Apache Kafka autogéré
- Utiliser un sujet Kafka comme destination en cas d'échec
- Résolution des erreurs de mappage des sources d’événements Kafka

Utilisation de Lambda avec Amazon MSK

Amazon Managed Streaming for Apache Kafka (Amazon MSK) est un service entièrement géré qui vous permet de créer et d'exécuter des applications qui utilisent Apache Kafka pour traiter des données en streaming. Amazon MSK simplifie la configuration, la mise à l'échelle et la gestion des clusters Kafka. Amazon MSK facilite également la configuration de votre application pour plusieurs zones de disponibilité et pour des raisons de sécurité avec Gestion des identités et des accès AWS (IAM).

Ce chapitre explique comment utiliser un cluster Amazon MSK en tant que source d'événements pour votre fonction Lambda. Le processus général d'intégration d'Amazon MSK à Lambda comprend les étapes suivantes :

1. Configuration du cluster et du réseau : configurez d'abord votre cluster Amazon MSK. Cela inclut la configuration réseau correcte pour permettre à Lambda d'accéder à votre cluster.
2. Configuration du mappage des sources d'événements : créez ensuite la ressource de mappage des sources d'événements dont Lambda a besoin pour connecter de façon sécurisée votre cluster Amazon MSK à votre fonction.
3. Configuration de la fonction et des autorisations : enfin, assurez-vous que votre fonction est correctement configurée et qu'elle dispose des autorisations nécessaires dans son rôle d'exécution.

> **Note**
>
> Vous pouvez désormais créer et gérer vos mappages de sources d'événements Amazon MSK directement depuis la console Lambda ou Amazon MSK. Les deux consoles offrent la possibilité de gérer automatiquement la configuration des autorisations de rôle d'exécution Lambda nécessaires pour un processus de configuration plus rationalisé.

Pour des exemples expliquant comment configurer une intégration Lambda avec un cluster Amazon MSK, consultez les sections the section called "didacticiel" Utilisation d'Amazon MSK comme source d'événements AWS Lambda sur le blog AWS Compute et Intégration d'Amazon MSK Lambda dans les Amazon MSK Labs.

Rubriques

- Exemple d'évènement

- Configuration de votre cluster Amazon MSK et de votre réseau Amazon VPC pour Lambda
- Configuration des autorisations Lambda pour les mappages de sources d'événements Amazon MSK
- Configuration de source d’événements Amazon MSK pour Lambda
- Tutoriel : Utilisation d’un mappage des sources d’événements Amazon MSK pour invoquer une fonction Lambda

Exemple d’évènement

Lambda envoie le lot de messages dans le paramètre d’événement quand il invoque votre fonction. La charge utile d’un événement contient un tableau de messages. Chaque élément de tableau contient des détails de la rubrique Amazon MSK et un identifiant de partition, ainsi qu’un horodatage et un message codé en base 64.


```
{
   "eventSource":"aws:kafka",
   "eventSourceArn":"arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
   "bootstrapServers":"b-2.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092",
   "records":{
      "mytopic-0":[
         {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "key":"abcDEFghiJKLmnoPQRstuVWXyz1234==",
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[
               {
                  "headerKey":[
                     104,
                     101,
                     97,
                     100,
                     101,
                     114,
```

```
                    86,
                    97,
                    108,
                    117,
                    101
                ]
            }
        ]
    }
  ]
 }
}
```

Configuration de votre cluster Amazon MSK et de votre réseau Amazon VPC pour Lambda

Pour connecter votre fonction AWS Lambda à votre cluster Amazon MSK, vous devez configurer correctement votre cluster et l'instance Amazon Virtual Private Cloud (VPC) dans laquelle il réside. Cette page décrit comment configurer votre cluster et votre VPC. Si votre cluster et votre VPC sont déjà correctement configurés, consultez the section called "Configurer une source d'événements" pour configurer le mappage des sources d'événements.

Rubriques

- Présentation des exigences de configuration réseau pour les intégrations Lambda et MSK
- Configuration d'une passerelle NAT pour une source d'événements MSK
- Configuration des points de terminaison AWS PrivateLink pour une source d'événements MSK

Présentation des exigences de configuration réseau pour les intégrations Lambda et MSK

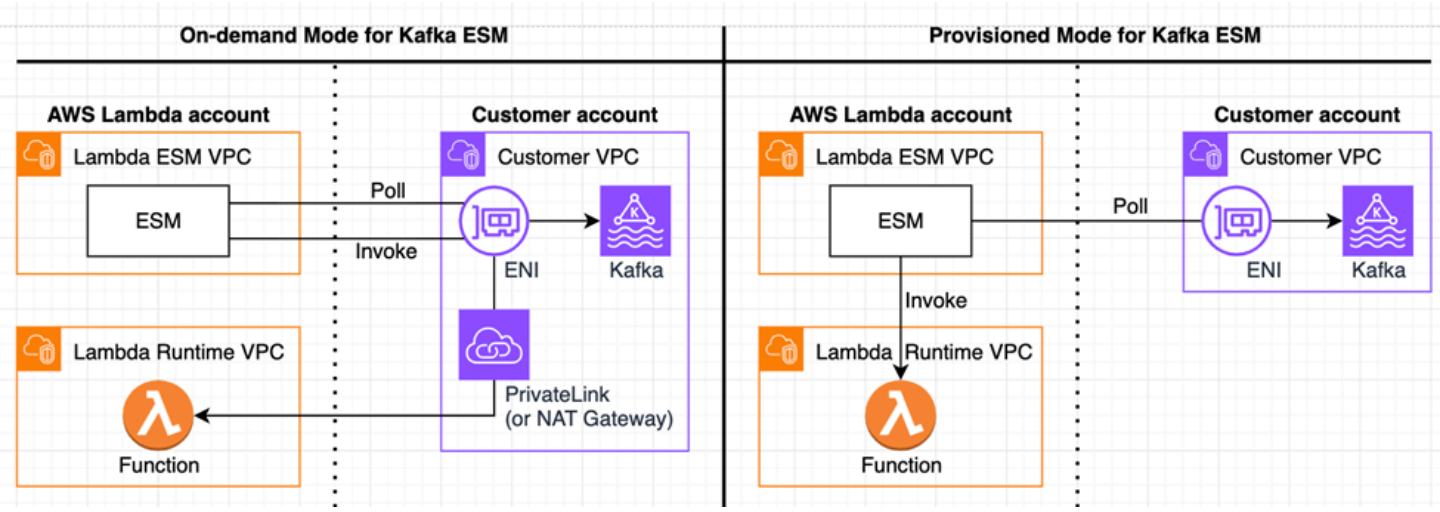
La configuration réseau requise pour une intégration Lambda et MSK dépend de l'architecture réseau de votre application. Trois ressources principales sont impliquées dans cette intégration : le cluster Amazon MSK, la fonction Lambda et le mappage des sources d'événements Lambda. Chacune de ces ressources réside dans un VPC différent :

- Votre cluster Amazon MSK réside généralement dans un sous-réseau privé d'un VPC que vous gérez.
- Votre fonction Lambda réside dans un VPC géré par AWS appartenant à Lambda.

- Votre mappage des sources d'événements Lambda réside dans un autre VPC géré par AWS appartenant à Lambda, distinct du VPC qui contient votre fonction.

Le mappage des sources d'événements est la ressource intermédiaire entre le cluster MSK et la fonction Lambda. Le mappage des sources d'événements effectue deux tâches principales. Tout d'abord, il interroge votre cluster MSK à la recherche de nouveaux messages. Ensuite, il invoque votre fonction Lambda avec ces messages. Étant donné que ces trois ressources se trouvent dans des VPC différents, les opérations d'interrogation et d'invocation nécessitent des appels réseau entre VPC.

Les exigences de configuration réseau pour votre mappage des sources d'événements varient selon qu'il utilise le mode provisionné ou le mode à la demande, comme le montre le schéma suivant :



La façon dont le mappage des sources d'événements Lambda interroge votre cluster MSK à la recherche de nouveaux messages est la même dans les deux modes. Pour établir une connexion entre votre mappage des sources d'événements et votre cluster MSK, Lambda crée une ENI Hyperplane (ou réutilise une ENI existante, si disponible) dans votre sous-réseau privé afin d'établir une connexion sécurisée. Comme illustré dans le schéma, cette ENI Hyperplane utilise la configuration du sous-réseau et du groupe de sécurité de votre cluster MSK, et non votre fonction Lambda.

Après avoir interrogé le message du cluster, la façon dont Lambda invoque votre fonction est différente dans chaque mode :

- En mode provisionné, Lambda gère automatiquement la connexion entre le VPC du mappage des sources d'événements et le VPC de la fonction. Vous n'avez donc pas besoin de composants de mise en réseau supplémentaires pour invoquer votre fonction avec succès.

- En mode à la demande, votre mappage des sources d'événements Lambda invoque votre fonction via un chemin passant par votre VPC géré par le client. Pour cette raison, vous devez configurer soit une passerelle NAT dans le sous-réseau public de votre VPC, soit des points de terminaison AWS PrivateLink dans le sous-réseau privé du VPC qui fournissent un accès à Lambda, AWS Security Token Service (STS) et éventuellement AWS Secrets Manager. La configuration correcte de l'une ou l'autre de ces options permet d'établir une connexion entre votre VPC et le VPC d'environnement d'exécution géré par Lambda, ce qui est nécessaire pour invoquer votre fonction.

Une passerelle NAT permet aux ressources de votre sous-réseau privé d'accéder à l'Internet public. L'utilisation de cette configuration signifie que votre trafic traverse Internet avant d'invoquer la fonction Lambda. Les points de terminaison AWS PrivateLink permettent aux sous-réseaux privés de se connecter en toute sécurité aux services AWS ou à d'autres ressources VPC privées sans passer par l'Internet public. Consultez the section called "Configuration d'une passerelle NAT pour une source d'événements MSK" ou the section called "Configuration des points de terminaison AWS PrivateLink pour une source d'événements MSK" pour en savoir plus sur la façon de configurer ces ressources.

Jusqu'à présent, nous avons supposé que votre cluster MSK résidait dans un sous-réseau privé au sein de votre VPC, ce qui est le cas le plus courant. Toutefois, même si votre cluster MSK se trouve dans un sous-réseau public au sein de votre VPC, vous devez configurer des points de terminaison AWS PrivateLink pour permettre une connexion sécurisée. Le tableau suivant résume les exigences de configuration réseau en fonction de la façon dont vous configurez votre cluster MSK et du mappage des sources d'événements Lambda :

Emplacement du cluster MSK (dans un VPC géré par le client)	Mode de mise à l'échelle du mappage des sources d'événements Lambda	Configuration réseau requise
Sous-réseau privé	Mode de capacité à la demande	Passerelle NAT (dans le sous-réseau public de votre VPC) ou points de terminaison AWS PrivateLink (dans le sous-réseau privé de votre VPC) pour permettre l'accès à Lambda, AWS STS, et éventuellement Secrets Manager.

Emplacement du cluster MSK (dans un VPC géré par le client)	Mode de mise à l'échelle du mappage des sources d'événements Lambda	Configuration réseau requise
Public subnet	Mode de capacité à la demande	Points de terminaison AWS PrivateLink (dans le sous-rése au public de votre VPC) pour permettre l'accès à Lambda, AWS STS, et éventuellement Secrets Manager.
Sous-réseau privé	Mode alloué	Aucune
Public subnet	Mode alloué	Aucune

En outre, les groupes de sécurité associés à votre cluster MSK doivent autoriser le trafic sur les bons ports. Vérifiez que les règles de groupe de sécurité suivantes sont configurées :

- Règles entrantes : autorisez tout le trafic sur le port d'agent par défaut. Le port utilisé par MSK dépend du type d'authentification sur le cluster : `9098` pour l'authentification IAM, `9096` pour SASL/SCRAM et `9094` pour TLS. Vous pouvez également utiliser une règle de groupe de sécurité avec référence circulaire pour autoriser l'accès à partir d'instances appartenant au même groupe de sécurité.
- Règles sortantes : autorisez tout le trafic sur le port `443` pour les destinations externes si votre fonction doit communiquer avec d'autres services AWS. Vous pouvez utiliser une règle de groupe de sécurité avec référence circulaire pour limiter l'accès à l'agent si vous n'avez pas besoin de communiquer avec d'autres services AWS.
- Règles entrantes de point de terminaison Amazon VPC :si vous utilisez un point de terminaison Amazon VPC, le groupe de sécurité associé à votre point de terminaison doit autoriser le trafic entrant sur le port `443` en provenance du groupe de sécurité du cluster.

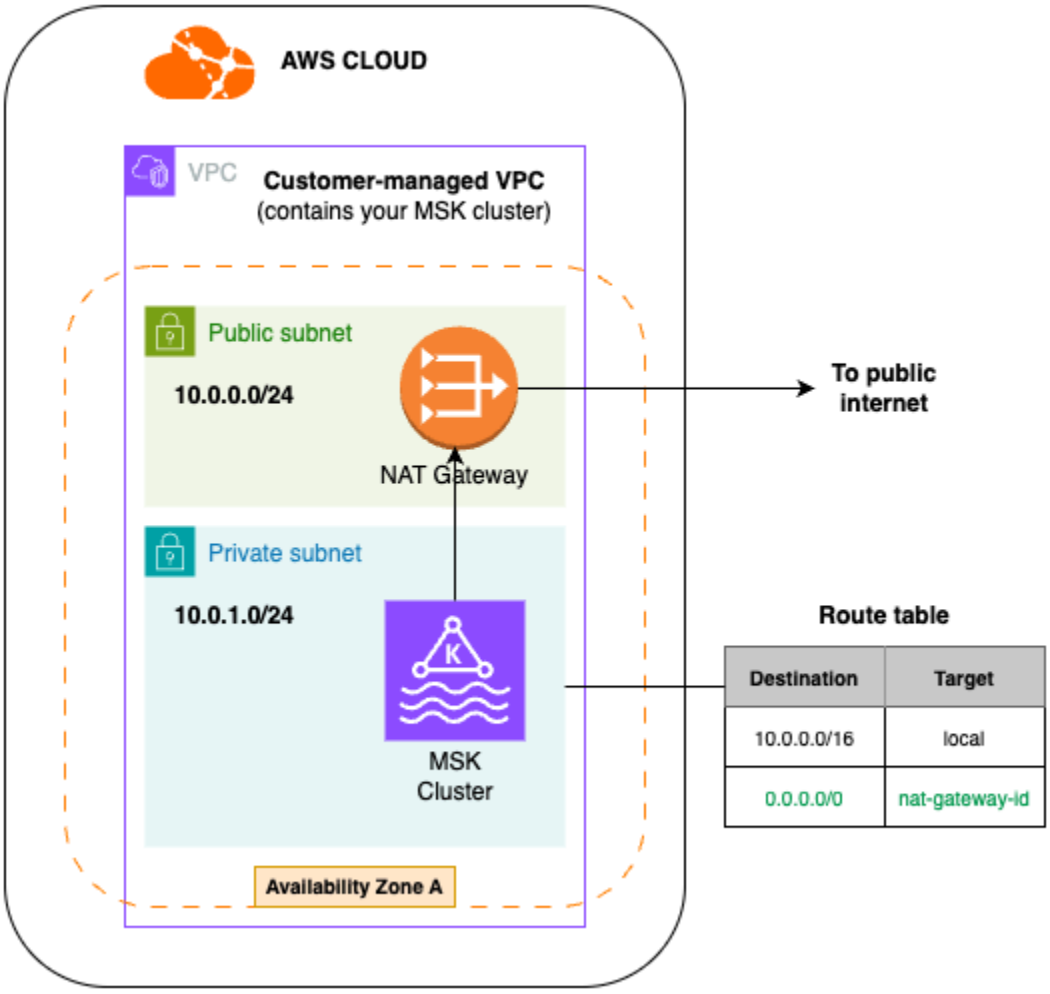
Configuration d'une passerelle NAT pour une source d'événements MSK

Vous pouvez configurer une passerelle NAT pour permettre à votre mappage des sources d'événements d'interroger les messages de votre cluster et d'invoquer la fonction via un chemin à travers votre VPC. Cela n'est nécessaire que si votre mappage des sources d'événements utilise

le mode à la demande et que votre cluster réside dans un sous-réseau privé de votre VPC. Si votre cluster réside dans un sous-réseau public de votre VPC, ou si le mappage des sources d'événements utilise le mode provisionné, vous n'avez pas besoin de configurer de passerelle NAT.

Une passerelle NAT permet aux ressources d'un sous-réseau privé d'accéder à l'Internet public. Si vous avez besoin d'une connectivité privée à Lambda, consultez plutôt the section called "Configuration des points de terminaison AWS PrivateLink pour une source d'événements MSK".

Après avoir configuré votre passerelle NAT, vous devez configurer les tables de routage appropriées. Cela permet au trafic provenant de votre sous-réseau privé d'être acheminé vers l'Internet public via la passerelle NAT.



Les étapes suivantes vous guident dans la mise en réseau d'une passerelle NAT à l'aide de la console. Si nécessaire, répétez ces étapes pour chaque zone de disponibilité (AZ).

Configurer une passerelle NAT et un routage approprié (console)

1. Suivez les étapes de Créer une passerelle NAT, en notant ce qui suit :
   - Les passerelles NAT doivent toujours résider dans un sous-réseau public. Créez des passerelles NAT avec une connectivité publique.
   - Si votre cluster MSK est répliqué à travers plusieurs AZ, créez une passerelle NAT par AZ. Par exemple, dans chaque AZ, votre VPC doit avoir un sous-réseau privé contenant votre cluster et un sous-réseau public contenant votre passerelle NAT. Pour une configuration avec trois AZ, vous aurez trois sous-réseaux privés, trois sous-réseaux publics et trois passerelles NAT.
2. Après avoir créé votre passerelle NAT, ouvrez la console Amazon VPC et choisissez Tables de routage dans le menu de gauche.
3. Choisissez Créer une table de routage.
4. Associez cette table de routage au VPC contenant votre cluster MSK. Vous pouvez éventuellement saisir un nom pour votre table de routage.
5. Choisissez Créer une table de routage.
6. Choisissez la table de routage que vous venez de créer.
7. Sur l'onglet Associations de sous-réseau, choisissez Modifier les associations de sous-réseau.
   - Associez cette table de routage au sous-réseau privé contenant votre cluster MSK.
8. Choisissez Edit routes (Modifier des routes).
9. Choisissez Ajouter une route :
   1. Pour Destination, choisissez `0.0.0.0/0`.
   2. Pour Cible, choisissez Passerelle NAT.
   3. Dans la zone de recherche, choisissez la passerelle NAT que vous avez créée à l'étape 1. Il doit s'agir de la passerelle NAT située dans la même AZ que le sous-réseau privé contenant votre cluster MSK (le sous-réseau privé que vous avez associé à cette table de routage à l'étape 6).
10. Sélectionnez Enregistrer les modifications.

Configuration des points de terminaison AWS PrivateLink pour une source d'événements MSK

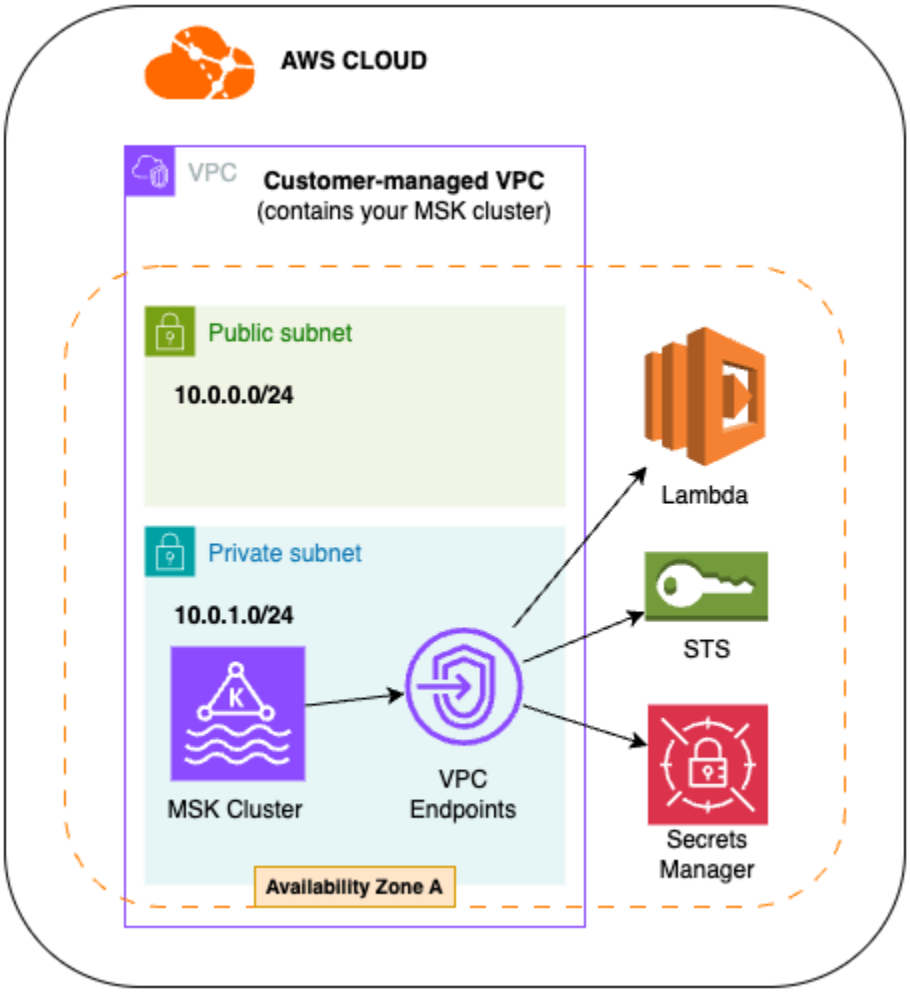
Vous pouvez configurer des points de terminaison AWS PrivateLink pour interroger les messages de votre cluster et invoquer la fonction via un chemin à travers votre VPC. Ces points de terminaison devraient permettre à votre cluster MSK d'accéder à ce qui suit :

- Le service Lambda
- AWS Security Token Service (STS)
- Éventuellement le service AWS Secrets Manager. Il est nécessaire si le secret requis pour l'authentification du cluster est stocké dans Secrets Manager.

La configuration des points de terminaison PrivateLink n'est requise que si votre mappage des sources d'événements utilise le mode à la demande. Si votre mappage des sources d'événements utilise le mode provisionné, Lambda établit les connexions requises pour vous.

Les points de terminaison PrivateLink permettent un accès privé et sécurisé aux services AWS sur AWS PrivateLink. Vous pouvez également configurer une passerelle NAT afin de permettre à votre cluster MSK d'accéder à l'Internet public (voir the section called "Configuration d'une passerelle NAT pour une source d'événements MSK").

Une fois que vous avez configuré vos points de terminaison de VPC, votre cluster MSK devrait disposer d'un accès direct et privé à Lambda, STS et éventuellement Secrets Manager.



Les étapes suivantes vous guident dans la configuration d'un point de terminaison PrivateLink à l'aide de la console. Si nécessaire, répétez ces étapes pour chaque point de terminaison (Lambda, STS, Secrets Manager).

Configurer des points de terminaison de VPC PrivateLink (console)

1. Ouvrez la console Amazon VPC et choisissez Points de terminaison dans le menu de gauche.
2. Choisissez Créer un point de terminaison.
3. Vous pouvez éventuellement saisir un nom pour votre point de terminaison.
4. Dans Type, sélectionnez Services AWS.
5. Sous Services, commencez à saisir le nom du service. Par exemple, pour créer un point de terminaison pour la connexion à Lambda, saisissez `lambda` dans le champ de recherche.

6. Dans les résultats, vous devriez voir le point de terminaison de service dans la région actuelle. Par exemple, dans la région USA Est (Virginie du Nord), vous devriez voir `com.amazonaws.us-east-2.lambda`. Sélectionnez ce service.
7. Sous Paramètres réseau, sélectionnez le VPC qui contient votre cluster MSK.
8. Sous Sous-réseaux, sélectionnez les AZ dans lesquelles se trouve votre cluster MSK.
   - Pour chaque AZ, sous ID de sous-réseau, choisissez le sous-réseau privé contenant votre cluster MSK.
9. Sous Groupes de sécurité, sélectionnez les groupes de sécurité associés à votre cluster MSK.
10. Choisissez Créer un point de terminaison.

Par défaut, les points de terminaison Amazon VPC disposent de politiques IAM ouvertes qui permettent un accès étendu aux ressources. La meilleure pratique consiste à restreindre ces politiques pour effectuer les actions nécessaires à l'aide de ce point de terminaison. Par exemple, pour votre point de terminaison Secrets Manager, vous pouvez modifier sa politique de sorte qu'il autorise uniquement le rôle d'exécution de votre fonction à accéder au secret.

Example Politique de point de terminaison de VPC – Point de terminaison Secrets Manager

```
{
    "Statement": [
        {
            "Action": "secretsmanager:GetSecretValue",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "arn:aws::iam::123456789012:role/my-role"
                ]
            },
            "Resource": "arn:aws::secretsmanager:us-west-2:123456789012:secret:my-
secret"
        }
    ]
}
```

Pour les points de terminaison AWS STS et Lambda, vous pouvez limiter le principal appelant au principal de service Lambda. Cependant, assurez-vous d'utiliser `"Resource": "*"` dans ces politiques.

Example Politique de point de terminaison de VPC – Point de terminaison AWS STS

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Example Politique de point de terminaison de VPC – Point de terminaison Lambda

```
{
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Configuration des autorisations Lambda pour les mappages de sources d'événements Amazon MSK

Pour accéder au cluster Amazon MSK, votre fonction et votre mappage des sources d'événements ont besoin d'autorisations pour effectuer diverses actions de l'API Amazon MSK. Ajoutez ces autorisations au rôle d'exécution de votre fonction. Si vos utilisateurs ont besoin de l'accès, ajoutez les autorisations requises à la stratégie d'identité de l'utilisateur ou du rôle correspondant.

La politique de gestion des AWSLambdaMSKExecutionrôles contient les autorisations minimales requises pour les mappages de sources d'événements Amazon MSK Lambda. Pour simplifier le processus d'autorisation, vous pouvez :

- Associez la politique de gestion des AWSLambdaMSKExecutionrôles à votre rôle d'exécution.
- Laissez la console Lambda générer les autorisations pour vous. Lorsque vous créez un mappage de source d'événement Amazon MSK dans la console, Lambda évalue votre rôle d'exécution et vous avertit si des autorisations sont manquantes. Choisissez Générer des autorisations pour mettre à jour automatiquement votre rôle d'exécution. Cela ne fonctionne pas si vous avez créé ou modifié manuellement vos politiques de rôle d'exécution, ou si les politiques sont associées à plusieurs rôles. Notez que des autorisations supplémentaires peuvent toujours être requises dans votre rôle d'exécution lorsque vous utilisez des fonctionnalités avancées telles que la destination en cas d'échec ou le registre des AWS Glue schémas.

Rubriques

- Autorisations requises
- Autorisations facultatives

Autorisations requises

Votre rôle d'exécution de fonction Lambda doit disposer des autorisations requises suivantes pour les mappages de sources d'événements Amazon MSK. Ces autorisations sont incluses dans la politique de gestion des AWSLambdaMSKExecutionrôles.

CloudWatch Autorisations de journalisation

Les autorisations suivantes permettent à Lambda de créer et de stocker des journaux dans Amazon CloudWatch Logs.

- journaux : CreateLogGroup
- journaux : CreateLogStream
- journaux : PutLogEvents

Autorisations du cluster MSK

Les autorisations suivantes permettent à Lambda d'accéder à votre cluster Amazon MSK en votre nom :

- Kafka : DescribeCluster
- Source : V2 DescribeCluster
- Kafka : GetBootstrapBrokers

Nous recommandons d'utiliser kafka : DescribeCluster V2 au lieu de kafka :. DescribeCluster L'autorisation v2 fonctionne à la fois avec les clusters Amazon MSK provisionnés et sans serveur. Vous n'avez besoin que d'une de ces autorisations dans votre politique.

Autorisations VPC

Les autorisations suivantes permettent à Lambda de créer et de gérer des interfaces réseau lors de la connexion à votre cluster Amazon MSK :

- EC2 : CreateNetworkInterface
- EC2 : DescribeNetworkInterfaces
- EC2 : DescribeVpcs
- EC2 : DeleteNetworkInterface
- EC2 : DescribeSubnets
- EC2 : DescribeSecurityGroups

Autorisations facultatives

Votre fonction Lambda peut également nécessiter ces autorisations pour :

- Accédez à des clusters Amazon MSK multicomptes. Pour les mappages de sources d'événements entre comptes, vous avez besoin de kafka : DescribeVpcConnection dans le rôle d'exécution. Un directeur IAM qui crée un mappage de sources d'événements entre comptes a besoin de Kafka :. ListVpcConnections
- Accédez à votre secret SCRAM si vous utilisez l'authentification SASL/SCRAM. Cela permet à votre fonction d'utiliser un nom d'utilisateur et un mot de passe pour se connecter à Kafka.
- Décrivez le secret de votre Secrets Manager, si vous utilisez l' SASL/SCRAM authentification mTLS. Cela permet à votre fonction de récupérer les informations d'identification ou les certificats nécessaires pour des connexions sécurisées.
- Accédez à votre clé gérée par le AWS KMS client, si votre AWS Secrets Manager secret est chiffré à l'aide d'une clé gérée par le AWS KMS client.

- Accédez aux secrets de votre registre de schémas, si vous utilisez un registre de schémas avec authentification :
  - Pour le registre des AWS Glue schémas : les besoins `glue:GetRegistry` et `glue:GetSchemaVersion` les autorisations de votre fonction. Elles permettent à votre fonction de rechercher et d'utiliser les règles de format de message stockées dans AWS Glue.
  - Pour le registre de schémas Confluent avec `BASIC_AUTH` ou `CLIENT_CERTIFICATE_TLS_AUTH` : votre fonction a besoin de l'autorisation `secretsmanager:GetSecretValue` pour accéder au secret contenant les informations d'authentification. Cela permet à votre fonction de récupérer les username/password certificats nécessaires pour accéder au registre des schémas Confluent.
  - Pour les certificats CA privés : votre fonction a besoin de l'GetSecretValue autorisation secretsmanager : pour le secret contenant le certificat. Cela permet à votre fonction de vérifier l'identité des registres de schémas qui utilisent des certificats personnalisés.
- Accédez aux groupes de consommateurs du cluster Kafka et interrogez les messages depuis le sujet, si vous utilisez l'authentification IAM pour le mappage des sources d'événements.

Cela correspond aux autorisations requises suivantes :

- kafka : ListScramSecrets - Permet de répertorier les secrets SCRAM pour l'authentification Kafka
- secretsmanager : GetSecretValue - Permet de récupérer des secrets depuis Secrets Manager
- KMS:Decrypt - Permet le déchiffrement de données cryptées en utilisant AWS KMS
- glue : GetRegistry - Autorise l'accès au registre des AWS Glue schémas
- glue : GetSchemaVersion - Permet de récupérer des versions de schéma spécifiques à partir du registre des AWS Glue schémas
- Kafka-Cluster:connect - Accorde l'autorisation de se connecter et de s'authentifier auprès du cluster
- kafka-cluster : AlterGroup - Accorde l'autorisation de rejoindre des groupes sur un cluster, équivalent à l'ACL READ GROUP d'Apache Kafka
- kafka-cluster : DescribeGroup - Accorde l'autorisation de décrire les groupes d'un cluster, équivalent à l'ACL DESCRIBE GROUP d'Apache Kafka
- kafka-cluster : DescribeTopic - Accorde l'autorisation de décrire les sujets d'un cluster, équivalent à l'ACL DESCRIBE TOPIC d'Apache Kafka
- kafka-cluster : ReadData - Accorde l'autorisation de lire les données des rubriques d'un cluster, équivalent à l'ACL READ TOPIC d'Apache Kafka

En outre, si vous souhaitez envoyer des enregistrements d'invocations ayant échoué vers une destination en cas d'échec, vous devez disposer des autorisations suivantes en fonction du type de destination :

- Pour les destinations Amazon SQS : sqs : SendMessage - Permet d'envoyer des messages à une file d'attente Amazon SQS
- Pour les destinations Amazon SNS : sns:Publish : permet de publier des messages sur une rubrique Amazon SNS
- Pour les destinations du compartiment Amazon S3 : s3 : PutObject et s3 : ListBucket - Permet d'écrire et de répertorier des objets dans un compartiment Amazon S3

Pour résoudre les erreurs d'authentification et d'autorisation, consultez the section called "Résolution des problèmes".

Configuration de source d'événements Amazon MSK pour Lambda

Pour utiliser un cluster Amazon MSK comme source d'événements pour votre fonction Lambda, vous devez créer un mappage des sources d'événements qui connecte les deux ressources. Cette page explique comment créer un mappage des sources d'événements pour Amazon MSK.

Cette page suppose que vous avez déjà correctement configuré votre cluster MSK et l'instance Amazon Virtual Private Cloud (VPC) dans lequel il réside. Si vous devez configurer votre cluster ou VPC, consultez the section called "Configuration du cluster et du réseau". Pour configurer le comportement des nouvelles tentatives afin de gérer les erreurs, consultezthe section called "Réessayer les configurations".

Rubriques

- Utilisation d'un cluster Amazon MSK en tant que source d'événement
- Configuration des méthodes d'authentification du cluster Amazon MSK dans Lambda
- Création d'un mappage des sources d'événements Lambda pour une source d'événements Amazon MSK
- Création de mappages de sources d'événements entre comptes dans Lambda
- Tous les paramètres de configuration des sources d'événements Amazon MSK dans Lambda

Utilisation d’un cluster Amazon MSK en tant que source d’événement

Lorsque vous ajoutez votre cluster Apache Kafka ou Amazon MSK comme déclencheur pour votre fonction Lambda, le cluster est utilisé comme source d’événement.

Lambda lit les données d'événements des rubriques Kafka que vous spécifiez `Topics` dans une CreateEventSourceMappingdemande, en fonction de la position de départ que vous spécifiez. Lorsque le traitement a réussi, votre rubrique Kafka est validée dans votre cluster Kafka.

Lambda lit les messages séquentiellement pour chaque partition de rubrique Kafka. Une seule charge utile Lambda peut contenir des messages provenant de plusieurs partitions. Lorsque d'autres enregistrements sont disponibles, Lambda continue de traiter les enregistrements par lots, en fonction de la BatchSize valeur que vous spécifiez dans une CreateEventSourceMappingdemande, jusqu'à ce que votre fonction aborde le sujet.

Après avoir traité chaque lot, Lambda valide les décalages des messages dans celui-ci. Si votre fonction renvoie une erreur pour l’un des messages d’un lot, Lambda réessaie le lot de messages complet jusqu’à ce que le traitement réussisse ou que les messages expirent. Vous pouvez envoyer les enregistrements qui échouent à toutes les tentatives vers une destination en cas de panne pour un traitement ultérieur.

> **Note**
>
> Alors que les fonctions Lambda ont généralement un délai d’expiration maximal de 15 minutes, les mappages des sources d’événement pour Amazon MSK, Apache Kafka autogéré, Amazon DocumentDB et Amazon MQ pour ActiveMQ et RabbitMQ ne prennent en charge que les fonctions dont le délai d’expiration maximal est de 14 minutes.

Configuration des méthodes d'authentification du cluster Amazon MSK dans Lambda

Lambda doit être autorisé à accéder à votre cluster Amazon MSK, à récupérer des enregistrements et à effectuer d’autres tâches. Amazon MSK prend en charge plusieurs méthodes d’authentification auprès de votre cluster MSK.

Méthodes d’authentification des clusters

- Accès non authentifié
- Authentification SASL/SCRAM

- Authentification TLS mutuelle
- Authentification IAM
- Comment Lambda choisit un courtier bootstrap

Accès non authentifié

Si aucun client n'accède au cluster via Internet, vous pouvez utiliser un accès non authentifié.

Authentification SASL/SCRAM

Lambda prend en charge l'authentification simple et l'authentification SASL/SCRAM (Security Layer/Salted Challenge Response Authentication Mechanism), avec la fonction de hachage SHA-512 et le chiffrement TLS (Transport Layer Security). Pour que Lambda se connecte au cluster, stockez les informations d'identification d'authentification (nom d'utilisateur et mot de passe) dans un secret Secrets Manager, et référencez ce secret lors de la configuration de votre mappage des sources d'événements.

Pour plus d'informations sur l'utilisation de Secrets Manager, consultez Authentification par informations d'identification avec Secrets Manager dans le Guide du développeur Amazon Managed Streaming for Apache Kafka.

> **Note**
>
> Amazon MSK ne prend pas en charge SASL/PLAIN l'authentification.

Authentification TLS mutuelle

Mutual TLS (mTLS) fournit une authentification bidirectionnelle entre le client et le serveur. Le client envoie un certificat au serveur pour que ce dernier vérifie le client. Le serveur envoie également un certificat au client pour que ce dernier vérifie le serveur.

Pour les intégrations Amazon MSK avec Lambda, votre cluster MSK agit en tant que serveur et Lambda en tant que client.

- Pour que Lambda puisse vérifier votre cluster MSK, vous devez configurer un certificat client en tant que secret dans Secrets Manager et référencer ce certificat dans votre configuration de mappage des sources d'événements. Le certificat client doit être signé par une autorité de certification (CA) située dans le magasin d'approbations du serveur.

- Le cluster MSK envoie également un certificat de serveur à Lambda. Le certificat du serveur doit être signé par une autorité de certification (CA) dans le AWS trust store.

Amazon MSK ne prend pas en charge les certificats de serveur autosignés. Tous les courtiers d'Amazon MSK utilisent des certificats publics signés par Amazon Trust Services CAs, auxquels Lambda fait confiance par défaut.

Configuration du secret mTLS

Le secret CLIENT_CERTIFICATE_TLS_AUTH nécessite un champ de certificat et un champ de clé privée. Pour une clé privée chiffrée, le secret nécessite un mot de passe de clé privée. Le certificat et la clé privée doivent être au format PEM.

> **Note**
>
> Lambda prend en charge les algorithmes de chiffrement par clé privée PBES1(mais pas PBES2).

Le champ de certificat doit contenir une liste de certificats, commençant par le certificat client, suivi de tous les certificats intermédiaires et se terminant par le certificat racine. Chaque certificat doit commencer sur une nouvelle ligne avec la structure suivante :

```
-----BEGIN CERTIFICATE-----
        <certificate contents>
-----END CERTIFICATE-----
```

Secrets Manager prend en charge les secrets jusqu'à 65 536 octets, ce qui offre suffisamment d'espace pour de longues chaînes de certificats.

La clé privée doit être au format PKCS #8, avec la structure suivante :

```
-----BEGIN PRIVATE KEY-----
         <private key contents>
-----END PRIVATE KEY-----
```

Pour une clé privée chiffrée, utilisez la structure suivante :

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
          <private key contents>
```

```
-----END ENCRYPTED PRIVATE KEY-----
```

L'exemple suivant affiche le contenu d'un secret pour l'authentification mTLS à l'aide d'une clé privée chiffrée. Pour une clé privée chiffrée, vous devez inclure le mot de passe de clé privée dans le secret.

```
{
 "privateKeyPassword": "testpassword",
 "certificate": "-----BEGIN CERTIFICATE-----
MIIE5DCCAsygAwIBAgIRAPJdwaFaNRrytHBto0j5BA0wDQYJKoZIhvcNAQELBQAw
...
j0Lh4/+1HfgyE2KlmII36dg4IMzNjAFEBZiCRoPimO40s1cRqtFHXoal0QQbIlxk
cmUuiAii9R0=
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIFgjCCA2qgAwIBAgIQdjNZd6uFf9hbNC5RdfmHrzANBgkqhkiG9w0BAQsFADBb
...
rQoiowbbk5wXCheYSANQIfTZ6weQTgiCHCCbuuMKNVS95FkXm0vqVD/YpXKwA/no
c8PH3PSoAaRwMMgOSA2ALJvbRz8mpg==
-----END CERTIFICATE-----",
 "privateKey": "-----BEGIN ENCRYPTED PRIVATE KEY-----
MIIFKzBVBgkqhkiG9w0BBQ0wSDAnBgkqhkiG9w0BBQwwGgQUiAFcK5hT/X7Kjmgp
...
QrSekqF+kWzmB6nAfSzgO9IaoAaytLvNgGTckWeUkWn/V0Ck+LdGUXzAC4RxZnoQ
zp2mwJn2NYB7AZ7+imp0azDZb+8YG2aUCiyqb6PnnA==
-----END ENCRYPTED PRIVATE KEY-----"
}
```

Pour plus d'informations sur le protocole mTLS pour Amazon MSK et des instructions sur la génération d'un certificat client, consultez Authentification de client TLS mutuel pour Amazon MSK dans le Guide du développeur Amazon Managed Streaming for Apache Kafka.

Authentification IAM

Vous pouvez utiliser Gestion des identités et des accès AWS (IAM) pour authentifier l'identité des clients qui se connectent au cluster MSK. Avec l'authentification IAM, Lambda s'appuie sur les autorisations du rôle d'exécution de votre fonction pour se connecter au cluster, récupérer des enregistrements et effectuer d'autres actions requises.??? Pour un exemple de politique contenant les autorisations nécessaires, consultez Créer des politiques d'autorisation pour le rôle IAM dans le Guide du développeur Amazon Managed Streaming pour Apache Kafka.

Si l'authentification IAM est active sur votre cluster MSK et que vous ne fournissez pas de secret, Lambda utilise automatiquement l'authentification IAM par défaut.

Pour de plus amples informations sur l'authentification IAM dans Amazon MSK, veuillez consulter Contrôle d'accès IAM.

Comment Lambda choisit un courtier bootstrap

Lambda choisit un courtier d'amorçage en fonction des méthodes d'authentification disponibles sur votre cluster, et si vous fournissez un secret pour l'authentification. Si vous fournissez un secret pour les mTLS ou SASL/SCRAM, Lambda choisit automatiquement cette méthode d'authentification. Si vous ne e faites pas, Lambda sélectionne la méthode d'authentification la plus puissante active sur votre cluster. Voici l'ordre de priorité dans lequel Lambda sélectionne un courtier, de l'autorisation la plus forte à la plus faible :

- MTLs (secret fourni pour les mTLS)
- SASL/SCRAM (secret provided for SASL/SCRAM)
- SASL IAM (aucun secret fourni et authentification IAM active)
- TLS non authentifié (aucun secret fourni et authentification IAM non active)
- Texte brut (aucun secret n'est fourni, et l'authentification IAM et le TLS non authentifié ne sont pas actifs)

> **Note**
>
> Si Lambda ne parvient pas à se connecter au type de courtier le plus sécurisé, Lambda n'essaie pas de se connecter à un type de courtier différent (plus faible). Si vous souhaitez que Lambda choisisse un type de courtier plus faible, désactivez toutes les méthodes d'authentification plus puissantes sur votre cluster.

Création d'un mappage des sources d'événements Lambda pour une source d'événements Amazon MSK

Pour créer un mappage des sources d'événements, vous pouvez utiliser la console Lambda, AWS Command Line Interface (CLI) ou un SDK AWS.

> **Note**
>
> Lorsque vous créez le mappage des sources d'événements, Lambda crée une ENI Hyperplane dans le sous-réseau privé qui contient votre cluster MSK, ce qui permet à

Lambda d'établir une connexion sécurisée. Cette ENI Hyperplane utilise la configuration du sous-réseau et du groupe de sécurité de votre cluster MSK, et non votre fonction Lambda.

Les étapes de console suivantes ajoutent un cluster Amazon MSK comme déclencheur pour votre fonction Lambda. En arrière-plan, cela crée une ressource de mappage des sources d'événements.

Pour ajouter un déclencheur Amazon MSK à votre fonction Lambda (console)

1. Ouvrez la page Fonction de la console Lambda.
2. Choisissez le nom de la fonction Lambda pour laquelle vous souhaitez ajouter un déclencheur Amazon MSK.
3. Sous Function overview (Vue d'ensemble de la fonction), choisissez Add trigger (Ajouter un déclencheur).
4. Sous Trigger configuration (Configuration du déclencheur), choisissez MSK.
5. Pour spécifier les détails de votre cluster Kafka, procédez comme suit :
    1. Pour MSK cluster (Cluster MSK), sélectionnez votre cluster.
    2. Dans Nom de la rubrique, donnez un nom à la rubrique Kafka dont vous souhaitez consommer les messages.
    3. Pour l'identifiant de groupe de consommateurs, entrez l'identifiant d'un groupe de consommateurs Kafka à rejoindre, le cas échéant. Pour de plus amples informations, veuillez consulter the section called "ID de groupe de consommateurs".
6. Pour Authentification du cluster, effectuez les configurations nécessaires. Pour plus d'informations sur l'authentification du cluster, consultez the section called "Authentification du cluster".
    - Activez Utiliser l'authentification si vous souhaitez que Lambda effectue l'authentification auprès de votre cluster MSK lors de l'établissement d'une connexion. L'authentification est recommandée.
    - Si vous utilisez l'authentification, dans Méthode d'authentification, choisissez la méthode d'authentification à utiliser.
    - Si vous utilisez l'authentification, pour Clé Secrets Manager, choisissez la clé Secrets Manager qui contient les informations d'identification d'authentification nécessaires pour accéder à votre cluster.
7. Sous Configuration des sondeurs d'événements, effectuez les configurations nécessaires.

- Choisissez Activer le déclencheur pour activer le déclencheur immédiatement après sa création.
- Choisissez si vous souhaitez Configurer le mode provisionné pour votre mappage des sources d'événements. Pour de plus amples informations, veuillez consulter the section called “Mise à l'échelle des sondeurs d'événements”.
  - Si vous configurez le mode provisionné, entrez une valeur pour Minimum Event Sollers, une valeur pour Maximum Event Sollers et une valeur facultative pour PollerGroupName spécifier le regroupement de plusieurs ESMs au sein d'un même VPC source d'événements.
- Dans Position de départ, choisissez la façon dont vous souhaitez que Lambda commence à lire à partir de votre flux. Pour de plus amples informations, veuillez consulter the section called “Positions des interrogations et des flux”.

8. Sous Traitement par lots, effectuez les configurations nécessaires. Pour plus d'informations sur la mise en lots, consultez the section called “Comportement de traitement par lots”.
   1. Pour Batch size (Taille de lot), entrez le nombre maximum d'enregistrements à recevoir dans un même lot.
   2. Pour Batch window, veuillez saisir le nombre maximal de secondes nécessaires à Lambda pour collecter des enregistrements avant d'invoquer la fonction.
9. Sous Filtrage, effectuez les configurations nécessaires. Pour plus d'informations sur le filtrage, veuillez consulter la rubrique the section called “Filtrage des événements”.
   - Pour Critères de filtre, ajoutez des définitions de critères de filtre pour déterminer s'il faut ou non traiter un événement.
10. Sous Gestion des échecs, effectuez les configurations nécessaires. Pour plus d'informations sur la gestion des échecs, consultez the section called “Conserver les appels ayant échoué”.
    - Pour Destination en cas d'échec, spécifiez l'ARN de votre destination en cas d'échec.
11. Pour Balises, saisissez les balises à associer à ce mappage des sources d'événements.
12. Pour créer le déclencheur, choisissez Add (Ajouter).

Vous pouvez également créer le mappage des sources d'événements à l'aide de la AWS CLI avec la create-event-source-mappingcommande. L'exemple suivant crée un mappage des sources d'événements pour associer la fonction Lambda `my-msk-function` à la rubrique `AWSKafkaTopic`, en commençant par le message `LATEST`. Cette commande utilise également

l'SourceAccessConfigurationobjet pour demander à Lambda d'utiliser l'authentification SASL/SCRAM lors de la connexion au cluster.

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-
fd1b-45ad-85dd-15b4a5a6247e-2 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
  --source-access-configurations '[{"Type": "SASL_SCRAM_512_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:111122223333:secret:my-secret"}]'
```

Si le cluster utilise l'authentification mTLS, incluez un SourceAccessConfigurationobjet qui le spécifie `CLIENT_CERTIFICATE_TLS_AUTH` et un ARN de clé Secrets Manager. En voici un exemple dans la commande suivante :

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-
fd1b-45ad-85dd-15b4a5a6247e-2 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
  --source-access-configurations '[{"Type": "CLIENT_CERTIFICATE_TLS_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:111122223333:secret:my-secret"}]'
```

Lorsque le cluster utilise l'authentification IAM, vous n'avez pas besoin d' SourceAccessConfigurationobjet. En voici un exemple dans la commande suivante :

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-
fd1b-45ad-85dd-15b4a5a6247e-2 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
```

Création de mappages de sources d’événements entre comptes dans Lambda

Vous pouvez utiliser la connectivité privée multi-VPC pour connecter une fonction Lambda à un cluster MSK provisionné dans un autre Compte AWS. La connectivité multi-VPC utilise AWS PrivateLink, ce qui permet de maintenir tout le trafic sur le AWS réseau.

> **Note**
>
> Vous ne pouvez pas créer de mappages de sources d'événements entre comptes pour les clusters MSK sans serveur.

Pour créer un mappage des sources d'événements entre comptes, vous devez d'abord configurer la connectivité multi-VPC pour le cluster MSK. Lorsque vous créez le mappage des sources d'événements, utilisez l'ARN de connexion VPC géré au lieu de l'ARN du cluster, comme indiqué dans les exemples suivants. Le CreateEventSourceMappingfonctionnement varie également en fonction du type d'authentification utilisé par le cluster MSK.

Example — Crée un mappage des sources d'événements entre comptes pour le cluster qui utilise l'authentification IAM

Lorsque le cluster utilise l'authentification basée sur les rôles IAM, vous n'avez pas besoin d'objet. SourceAccessConfiguration Exemple :

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:vpc-connection/444455556666/
my-cluster-name/51jn98b4-0a61-46cc-b0a6-61g9a3d797d5-7 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
```

Example — Créez un mappage des sources d'événements entre comptes pour le cluster qui utilise l'authentification SASL/SCRAM

Si le cluster utilise l'authentification SASL/SCRAM, vous devez inclure un SourceAccessConfigurationobjet qui spécifie `SASL_SCRAM_512_AUTH` et un ARN secret du Secrets Manager.

Il existe deux manières d'utiliser les secrets pour les mappages de sources d'événements Amazon MSK entre comptes avec authentification : SASL/SCRAM

- Créez un secret dans le compte de fonction Lambda et synchronisez-le avec le secret du cluster. Créez une rotation pour synchroniser les deux secrets. Cette option vous permet de contrôler le secret depuis le compte de fonction.

- Utilisez le secret associé au cluster MSK. Ce secret doit autoriser l'accès intercompte au compte de la fonction Lambda. Pour plus d'informations, voir Autorisations d'accéder aux secrets AWS Secrets Manager pour les utilisateurs d'un autre compte.

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:vpc-connection/444455556666/
my-cluster-name/51jn98b4-0a61-46cc-b0a6-61g9a3d797d5-7 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function \
  --source-access-configurations '[{"Type": "SASL_SCRAM_512_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:444455556666:secret:my-secret"}]'
```

Example — Crée un mappage des sources d'événements entre comptes pour le cluster qui utilise l'authentification mTLS

Si le cluster utilise l'authentification mTLS, vous devez inclure un SourceAccessConfigurationobjet qui spécifie `CLIENT_CERTIFICATE_TLS_AUTH` et un ARN secret de Secrets Manager. Le secret peut être stocké dans le compte du cluster ou dans le compte de la fonction Lambda.

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:vpc-connection/444455556666/
my-cluster-name/51jn98b4-0a61-46cc-b0a6-61g9a3d797d5-7 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function \
  --source-access-configurations '[{"Type": "CLIENT_CERTIFICATE_TLS_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:444455556666:secret:my-secret"}]'
```

Tous les paramètres de configuration des sources d'événements Amazon MSK dans Lambda

Tous les types de sources d'événements Lambda partagent les mêmes opérations CreateEventSourceMappinget les mêmes opérations d'UpdateEventSourceMappingAPI. Cependant, seuls certains paramètres s'appliquent à Amazon MSK, comme indiqué dans le tableau suivant.

Paramètre	Obligatoire	Par défaut	Remarques
AmazonManagedKafkaEventSourceConfig	N	Contient le ConsumerGroupId	Peut définir uniquement sur Create (Créer)

Paramètre	Obligatoire	Par défaut	Remarques
		champ, dont la valeur par défaut est unique.	
BatchSize	N	100	Maximum : 10 000.
DestinationConfig	N	N/A	the section called “Conserver les appels ayant échoué”
Activé	N	True	
BisectBatchOnFunctionError	N	False	the section called “Réessayer les configurations”
FunctionResponseTypes	N	N/A	the section called “Réessayer les configurations”
MaximumRecordAgeInSeconds	N	-1 (infini)	the section called “Réessayer les configurations”
MaximumRetryAttempts	N	-1 (infini)	the section called “Réessayer les configurations”
EventSourceArn	Y	N/A	Peut définir uniquement sur Create (Créer)
FilterCriteria	N	N/A	Contrôle des événements envoyés par Lambda à votre fonction
FunctionName	Y	N/A	

Paramètre	Obligatoire	Par défaut	Remarques
KMSKeyArn	N	N/A	the section called “Chiffrement des critères de filtre”
MaximumBatchingWindowInSeconds	N	500 ms	Comportement de traitement par lots
ProvisionedPollersConfig	N	`MinimumPollers` : la valeur par défaut, si elle n’est pas spécifiée, est de 1 `MaximumPollers` : la valeur par défaut, si elle n’est pas spécifiée, est de 200 `PollerGroupName` : N/A	the section called “Mode alloué”
SourceAccessConfigurations	N	Pas d’informations d’identification	Informations d’identification d’authentification SASL/SCRAM ou CLIENT_CERTIFICATE_TLS_AUTH (TLS mutuel) pour votre source d’événement
StartingPosition	Y	N/A	AT_TIMESTAMP, TRIM_HORIZON ou DERNIER Peut définir uniquement sur Create (Créer)

Paramètre	Obligatoire	Par défaut	Remarques
StartingPositionTimestamp	N	N/A	Obligatoire s'il StartingPosition est défini sur AT_TIMESTAMP
Étiquettes	N	N/A	the section called “Balises de mappage des sources d’événements”
Rubriques	Y	N/A	Nom de rubrique Kafka Peut définir uniquement sur Create (Créer)

> **Note**
>
> Lorsque vous spécifiez un`PollerGroupName`, plusieurs ESMs au sein d'un même Amazon VPC peuvent partager la capacité de l'Event Poller Unit (EPU). Vous pouvez utiliser cette option pour optimiser les coûts du mode provisionné pour votre ESMs. Exigences relatives au regroupement ESM :
>
> - ESMs doit se trouver dans le même Amazon VPC
> - Maximum de 100 ESMs par groupe de sondeurs
> - Le nombre maximum de sondeurs agrégés pour l'ensemble ESMs d'un groupe ne peut pas dépasser 2 000
>
> Vous pouvez le mettre `PollerGroupName` à jour pour déplacer un ESM vers un autre groupe, ou supprimer un ESM d'un groupe en le définissant `PollerGroupName` sur une chaîne vide (« »).

Tutoriel : Utilisation d’un mappage des sources d’événements Amazon MSK pour invoquer une fonction Lambda

Dans ce tutoriel, vous exécuterez les étapes suivantes :

- Créez une fonction Lambda dans le même AWS compte qu'un cluster Amazon MSK existant.
- Configurer le réseau et l’authentification pour que Lambda communique avec Amazon MSK.
- Configurer un mappage des sources d’événements Amazon MSK Lambda, qui exécute votre fonction Lambda lorsque des événements apparaissent dans la rubrique.

Une fois ces étapes terminées, vous pouvez configurer une fonction Lambda pour traiter automatiquement les événements envoyés à Amazon MSK avec votre code Lambda personnalisé.

Que pouvez-vous faire avec cette fonctionnalité ?

Exemple de solution : Utiliser un mappage des sources d’événements MSK pour fournir des résultats en direct à vos clients.

Imaginons le scénario suivant : votre entreprise héberge une application Web dans laquelle vos clients peuvent consulter des informations sur des événements en direct, tels que des matchs de sport. Les informations actualisées du jeu sont fournies à votre équipe via une rubrique Kafka sur Amazon MSK. Vous souhaitez concevoir une solution qui utilise les mises à jour issues de la rubrique MSK afin de fournir une vue actualisée de l’événement en direct aux clients au sein d’une application que vous développez. Vous avez opté pour l’approche de conception suivante : vos applications clientes communiqueront avec un dorsal sans serveur hébergé dans AWS. Les clients se connecteront via des sessions websocket à l'aide de l'API Amazon WebSocket API Gateway.

Dans cette solution, vous avez besoin d’un composant qui lit les événements MSK, exécute une logique personnalisée pour préparer ces événements pour la couche application, puis transmet ces informations à l’API API Gateway. Vous pouvez implémenter ce composant en fournissant votre logique personnalisée dans une fonction Lambda, puis en l'appelant à l'aide d'un mappage de source d'événements AWS Lambda Amazon MSK. AWS Lambda

Pour plus d'informations sur la mise en œuvre de solutions à l'aide de l'API Amazon WebSocket API Gateway, consultez les WebSocket didacticiels sur les API dans la documentation d'API Gateway.

Conditions préalables

Un AWS compte avec les ressources préconfigurées suivantes :

Pour remplir ces prérequis, nous vous recommandons de suivre Get started using Amazon MSK dans la documentation Amazon MSK.

- Un cluster Amazon MSK. Consultez Create an Amazon MSK cluster dans Getting started using Amazon MSK.
- La configuration suivante :
  - Assurez-vous que l'authentification basée sur les rôles IAM est Activée dans les paramètres de sécurité de votre cluster. Cela améliore votre sécurité en limitant votre fonction Lambda à l'accès aux ressources Amazon MSK nécessaires uniquement. L'authentification basée sur les rôles IAM est activée par défaut sur les nouveaux clusters Amazon MSK.
  - Assurez-vous que l'Accès public est désactivé dans les paramètres réseau de votre cluster. Restreindre l'accès à Internet de votre cluster Amazon MSK améliore votre sécurité en limitant le nombre d'intermédiaires qui traitent vos données. L'accès public est activé par défaut sur les nouveaux clusters Amazon MSK.
- Une rubrique Kafka dans votre cluster Amazon MSK à utiliser pour cette solution. Consultez Create a topic dans Getting started using Amazon MSK.
- Un hôte administrateur Kafka configuré pour récupérer les informations de votre cluster Kafka et envoyer des événements Kafka à votre sujet à des fins de test, par exemple une EC2 instance Amazon sur laquelle la CLI d'administration Kafka et la bibliothèque Amazon MSK IAM sont installées. Consultez Create a client machine dans Getting started using Amazon MSK.

Une fois que vous avez configuré ces ressources, collectez les informations suivantes à partir de votre AWS compte pour confirmer que vous êtes prêt à continuer.

- Le nom de votre cluster Amazon MSK. Vous pouvez trouver cette information dans la console Amazon MSK.
- L'UUID du cluster, qui fait partie de l'ARN de votre cluster Amazon MSK, que vous pouvez trouver dans la console Amazon MSK. Suivez les procédures décrites dans la rubrique Listing clusters de la documentation Amazon MSK pour trouver cette information.
- Les groupes de sécurité associés à votre cluster Amazon MSK. Vous pouvez trouver cette information dans la console Amazon MSK. Dans les étapes suivantes, appelez-les vos*clusterSecurityGroups*.
- L'ID du VPC Amazon contenant votre cluster Amazon MSK. Vous pouvez trouver cette information en identifiant les sous-réseaux associés à votre cluster Amazon MSK dans la console Amazon MSK, puis en identifiant le VPC Amazon associé au sous-réseau dans la console Amazon VPC.

- Le nom de la rubrique Kafka utilisée dans votre solution. Vous pouvez trouver cette information en appelant votre cluster Amazon MSK à l'aide de la CLI `topics` Kafka depuis votre hôte administrateur Kafka. Pour plus d'informations sur la CLI de rubriques, consultez la section Adding and removing topics dans la documentation Kafka.
- Le nom d'un groupe de consommateurs pour votre rubrique Kafka, adapté à une utilisation par votre fonction Lambda. Ce groupe peut être créé automatiquement par Lambda. Vous n'avez donc pas besoin de le créer avec la CLI Kafka. Si vous devez gérer vos groupes de consommateurs, pour en savoir plus sur la CLI de groupes de consommateurs, consultez la rubrique Managing Consumer Groups dans la documentation Kafka.

Les autorisations suivantes dans votre AWS compte :

- L'autorisation de créer et de gérer une fonction Lambda.
- L'autorisation de créer des politiques IAM et de les associer à votre fonction Lambda.
- L'autorisation de créer des points de terminaison de VPC Amazon et de modifier la configuration réseau dans le VPC Amazon hébergeant votre cluster Amazon MSK.

Installez le AWS Command Line Interface

Si vous ne l'avez pas encore installé AWS Command Line Interface, suivez les étapes décrites dans la section Installation ou mise à jour de la dernière version du AWS CLI pour l'installer.

Ce tutoriel nécessite un terminal de ligne de commande ou un shell pour exécuter les commandes. Sous Linux et macOS, utilisez votre gestionnaire de shell et de package préféré.

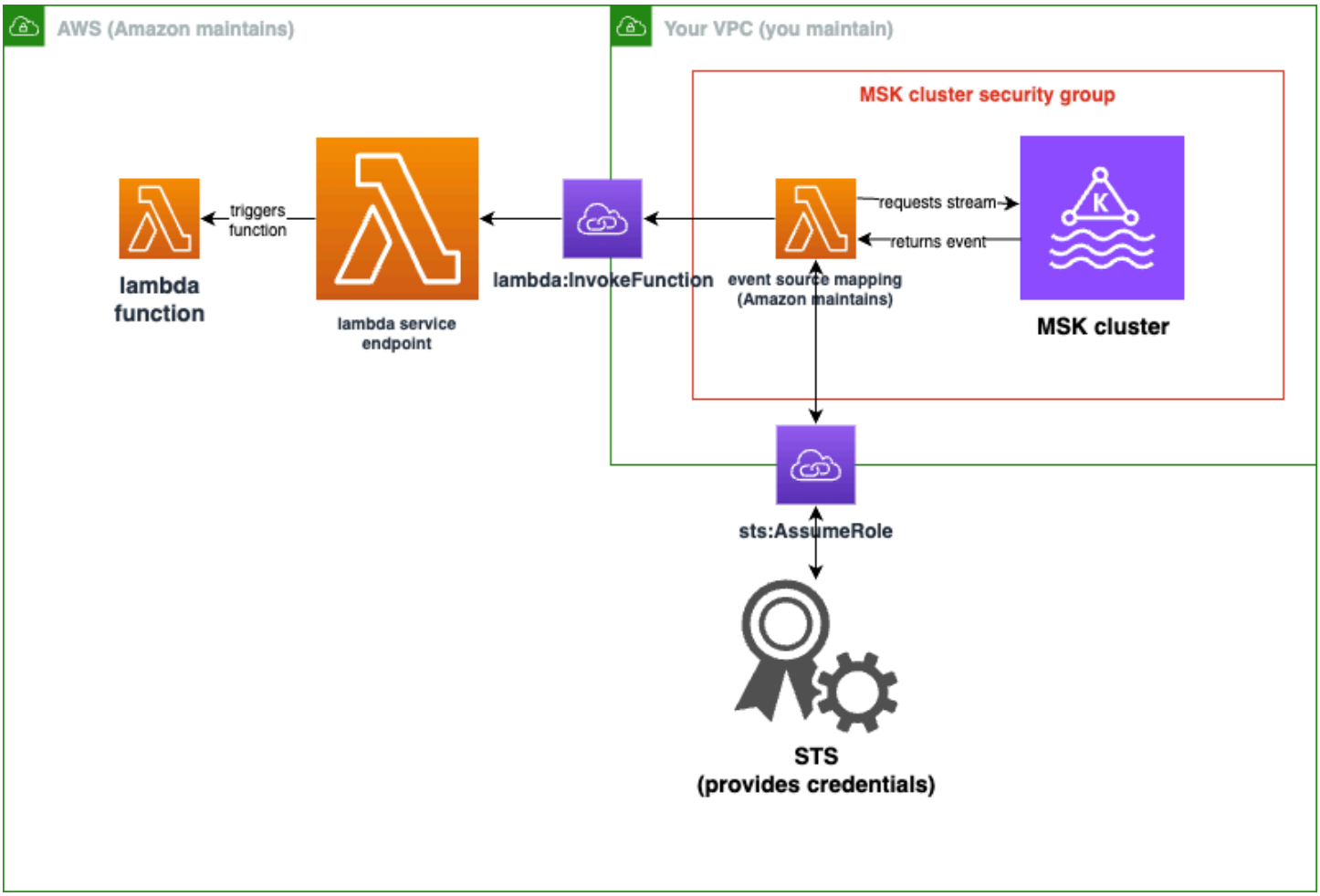
> **Note**
>
> Sous Windows, certaines commandes CLI Bash que vous utilisez couramment avec Lambda (par exemple `zip`) ne sont pas prises en charge par les terminaux intégrés du système d'exploitation. Installez le sous-système Windows pour Linux afin d'obtenir une version intégrée à Windows d'Ubuntu et Bash.

Configurer la connectivité réseau pour que Lambda communique avec Amazon MSK

AWS PrivateLink À utiliser pour connecter Lambda et Amazon MSK. Pour ce faire, vous créez des points de terminaison de VPC Amazon d'interface dans la console Amazon VPC. Pour plus

d'informations sur la configuration réseau, consultez the section called "Configuration du cluster et du réseau".

Lorsqu'un mappage des sources d'événements Amazon MSK s'exécute pour le compte d'une fonction Lambda, il endosse le rôle d'exécution de la fonction Lambda. Ce rôle IAM autorise le mappage pour accéder aux ressources sécurisées par IAM, telles que votre cluster Amazon MSK. Bien que les composants partagent un rôle d'exécution, le mappage Amazon MSK et votre fonction Lambda ont des exigences de connectivité distinctes pour leurs tâches respectives, comme le montre le schéma suivant.



Votre mappage des sources d'événements appartient au groupe de sécurité de votre cluster Amazon MSK. Au cours de cette étape de mise en réseau, créez des points de terminaison de VPC Amazon à partir de votre VPC de cluster Amazon MSK pour connecter le mappage des sources d'événements aux services Lambda et STS. Sécurisez ces points de terminaison pour accepter le trafic provenant du groupe de sécurité de votre cluster Amazon MSK. Ajustez ensuite les groupes de sécurité du cluster Amazon MSK pour permettre au mappage des sources d'événements de communiquer avec le cluster Amazon MSK.

Vous pouvez configurer les étapes suivantes à l'aide de l' AWS Management Console.

Pour configurer les points de terminaison de VPC Amazon d'interface afin de connecter Lambda et Amazon MSK

1. Créez un groupe de sécurité pour les points de terminaison Amazon VPC de votre interface*endpointSecurityGroup*, qui autorise le trafic TCP entrant sur 443 depuis. *clusterSecurityGroups* Suivez la procédure décrite dans Créer un groupe de sécurité dans la EC2 documentation Amazon pour créer un groupe de sécurité. Suivez ensuite la procédure décrite dans Ajouter des règles à un groupe de sécurité dans la EC2 documentation Amazon pour ajouter les règles appropriées.

   Créez un groupe de sécurité avec les informations suivantes :

   Lorsque vous ajoutez vos règles de trafic entrant, créez une règle pour chaque groupe de sécurité dans*clusterSecurityGroups*. Pour chaque règle :

   - Dans le champ Type, sélectionnez HTTPS.
   - Pour Source, sélectionnez l'un des*clusterSecurityGroups*.
2. Créez un point de terminaison connectant le service Lambda au VPC Amazon qui contient votre cluster Amazon MSK. Suivez la procédure décrite dans Create an interface endpoint.

   Créez un point de terminaison d'interface avec les informations suivantes :

   - Dans Nom du service`com.amazonaws.`*regionName*`.lambda`, sélectionnez où *regionName* héberge votre fonction Lambda.
   - Pour VPC, sélectionnez le VPC Amazon contenant votre cluster Amazon MSK.
   - Pour les groupes de sécurité*endpointSecurityGroup*, sélectionnez ceux que vous avez créés précédemment.
   - Pour Sous-réseaux, sélectionnez les sous-réseaux qui hébergent votre cluster Amazon MSK.
   - Pour Politique, fournissez le document de politique suivant, qui sécurise le point de terminaison afin qu'il soit utilisé par le principal de service Lambda pour l'action `lambda:InvokeFunction`.

   ```
   {
       "Statement": [
           {
               "Action": "lambda:InvokeFunction",
   ```

```
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

- Assurez-vous que Activer le nom DNS reste défini.

3. Créez un point de terminaison connectant le AWS STS service au Amazon VPC contenant votre cluster Amazon MSK. Suivez la procédure décrite dans Create an interface endpoint.

   Créez un point de terminaison d'interface avec les informations suivantes :

   - Pour Nom du service, sélectionnez AWS STS.
   - Pour VPC, sélectionnez le VPC Amazon contenant votre cluster Amazon MSK.
   - Pour les groupes de sécurité, sélectionnez*endpointSecurityGroup*.
   - Pour Sous-réseaux, sélectionnez les sous-réseaux qui hébergent votre cluster Amazon MSK.
   - Pour Politique, fournissez le document de politique suivant, qui sécurise le point de terminaison afin qu'il soit utilisé par le principal de service Lambda pour l'action `sts:AssumeRole`.

   ```
   {
       "Statement": [
           {
               "Action": "sts:AssumeRole",
               "Effect": "Allow",
               "Principal": {
                   "Service": [
                       "lambda.amazonaws.com"
                   ]
               },
               "Resource": "*"
           }
       ]
   }
   ```

   - Assurez-vous que Activer le nom DNS reste défini.

4. Pour chaque groupe de sécurité associé à votre cluster Amazon MSK, c'est-à-dire dans*clusterSecurityGroups*, autorisez ce qui suit :

   - Autorisez tout le trafic TCP entrant et sortant sur le 9098 à tous*clusterSecurityGroups*, y compris à l'intérieur de celui-ci.
   - Autorisez tout le trafic TCP sortant sur le port 443.

   Une partie de ce trafic est autorisée par les règles des groupes de sécurité par défaut. Par conséquent, si votre cluster est attaché à un seul groupe de sécurité et que ce groupe possède des règles par défaut, il n'est pas nécessaire d'ajouter des règles. Pour ajuster les règles des groupes de sécurité, suivez les procédures décrites dans Ajouter des règles à un groupe de sécurité dans la EC2 documentation Amazon.

   Ajoutez des règles à vos groupes de sécurité avec les informations suivantes :

   - Pour chaque règle entrante ou sortante pour le port 9098, indiquez
     - Pour Type, sélectionnez Custom TCP (TCP personnalisé).
     - Pour Plage de ports, indiquez 9098.
     - Pour Source, indiquez l'un des*clusterSecurityGroups*.
   - Pour chaque règle entrante pour le port 443, pour Type, sélectionnez HTTPS.

Créer un rôle IAM pour que Lambda puisse lire un extrait de votre rubrique Amazon MSK

Identifiez les exigences d'authentification que Lambda doit lire dans votre rubrique Amazon MSK, puis définissez-les dans une politique. Créez un rôle qui autorise Lambda à utiliser ces autorisations. *lambdaAuthRole* Autorisez les actions sur votre cluster Amazon MSK à l'aide d'actions IAM `kafka-cluster`. Autorisez ensuite Lambda à effectuer les EC2 actions Amazon MSK et `kafka` Amazon nécessaires pour découvrir et se connecter à votre cluster Amazon MSK, ainsi que les actions permettant à Lambda de CloudWatch consigner ce qu'il a fait.

Pour décrire les exigences d'authentification pour que Lambda puisse lire depuis Amazon MSK

1. Rédigez un document de politique IAM (un document JSON) qui permet à Lambda de lire un extrait de votre sujet Kafka dans votre cluster Amazon MSK en utilisant votre groupe de consommateurs Kafka. *clusterAuthPolicy* Lambda nécessite qu'un groupe de consommateurs Kafka soit défini lors de la lecture.

Modifiez le modèle suivant pour l'aligner sur vos prérequis :

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kafka-cluster:Connect",
                "kafka-cluster:DescribeGroup",
                "kafka-cluster:AlterGroup",
                "kafka-cluster:DescribeTopic",
                "kafka-cluster:ReadData",
                "kafka-cluster:DescribeClusterDynamicConfiguration"
            ],
            "Resource": [
                "arn:aws:kafka:us-
east-1:111122223333:cluster/mskClusterName/cluster-uuid",
                "arn:aws:kafka:us-
east-1:111122223333:topic/mskClusterName/cluster-uuid/mskTopicName",
                "arn:aws:kafka:us-
east-1:111122223333:group/mskClusterName/cluster-uuid/mskGroupName"
            ]
        }
    ]
}
```

Pour plus d'informations, consultez the section called "Configurer les autorisations ". Lorsque vous rédigez votre politique :

- Remplacez *us-east-1* et *111122223333* par le Région AWS et Compte AWS de votre cluster Amazon MSK.
- Pour*mskClusterName*, indiquez le nom de votre cluster Amazon MSK.
- Pour*cluster-uuid*, fournissez l'UUID dans l'ARN de votre cluster Amazon MSK.
- Pour*mskTopicName*, indiquez le nom de votre sujet Kafka.
- Pour*mskGroupName*, indiquez le nom de votre groupe de consommateurs Kafka.

2. Identifiez Amazon MSK, Amazon EC2 et CloudWatch les autorisations requises pour que Lambda découvre et connecte votre cluster Amazon MSK, puis enregistrez ces événements.

   La politique gérée `AWSLambdaMSKExecutionRole` définit de manière permissive les autorisations requises. Utilisez-la dans les étapes suivantes.

   Dans un environnement de production, évaluez `AWSLambdaMSKExecutionRole` pour restreindre votre politique de rôle d'exécution sur la base du principe du moindre privilège, puis rédigez une politique pour votre rôle qui remplace cette politique gérée.

Pour plus d'informations sur le langage de politique IAM, consultez la documentation IAM.

Maintenant que vous avez rédigé votre document de politique, créez une politique IAM afin de pouvoir l'attacher à votre rôle. Pour ce faire, vous pouvez utiliser la console en suivant la procédure ci-dessous.

Pour créer une politique IAM à partir de votre document de politique

1. Connectez-vous à la console IAM AWS Management Console et ouvrez-la à https://console.aws.amazon.com/iam/l'adresse.
2. Dans le panneau de navigation de gauche, choisissez Politiques.
3. Choisissez Create Policy (Créer une politique).
4. Dans la section Éditeur de politiques, choisissez l'option JSON.
5. Coller*`clusterAuthPolicy`*.
6. Lorsque vous avez fini d'ajouter des autorisations à la politique, choisissez Suivant.
7. Sur la page Vérifier et créer, tapez un Nom de politique et une Description (facultative) pour la politique que vous créez. Vérifiez les Autorisations définies dans cette politique pour voir les autorisations accordées par votre politique.
8. Choisissez Create policy (Créer une politique) pour enregistrer votre nouvelle politique.

Pour plus d'informations, consultez Création de politiques IAM dans la documentation IAM.

Maintenant que vous disposez des politiques IAM appropriées, créez un rôle et attachez-les à celui-ci. Pour ce faire, vous pouvez utiliser la console en suivant la procédure ci-dessous.

Pour créer un rôle d'exécution dans la console IAM

1. Ouvrez la page Roles (Rôles) dans la console IAM.
2. Choisissez Créer un rôle.
3. Sous Trusted entity type (Type d'entité approuvée), choisissez service AWS .
4. Sous Cas d'utilisation, choisissez Lambda.
5. Choisissez Suivant.
6. Sélectionnez les stratégies suivantes :
   - *clusterAuthPolicy*
   - AWSLambdaMSKExecutionRole
7. Choisissez Suivant.
8. Dans Nom du rôle, entrez *lambdaAuthRole* puis choisissez Créer un rôle.

Pour de plus amples informations, veuillez consulter the section called "Rôle d'exécution (autorisations pour les fonctions d'accéder à d'autres ressources)".

Créer une fonction Lambda pour lire à partir de votre rubrique Amazon MSK

Créez une fonction Lambda configurée pour utiliser votre rôle IAM. Vous pouvez enregistrer votre fonction Lambda à l'aide de la console.

Pour créer une fonction Lambda à l'aide de votre configuration d'authentification

1. Ouvrez la console Lambda et choisissez Créer une fonction dans l'en-tête.
2. Sélectionnez Créer à partir de zéro.
3. Pour Nom de la fonction, saisissez un nom approprié de votre choix.
4. Pour Environnement d'exécution, choisissez la dernière version prise en charge (Dernier pris en charge) de `Node.js` pour utiliser le code fourni dans ce tutoriel.
5. Choisissez Modifier le rôle d'exécution par défaut.
6. Sélectionnez Utiliser un rôle existant.
7. Pour Rôle existant, sélectionnez*lambdaAuthRole*.

Dans un environnement de production, vous devez généralement ajouter des politiques supplémentaires au rôle d'exécution de votre fonction Lambda afin de traiter intelligemment vos

événements Amazon MSK. Pour plus d'informations sur l'ajout de politiques à votre rôle, consultez la section Ajouter ou supprimer des autorisations d'identité dans la documentation IAM.

Création d'un mappage des sources d'événements pour votre fonction Lambda

Votre mappage des sources d'événements Amazon MSK fournit au service Lambda les informations nécessaires pour invoquer votre fonction Lambda lorsque des événements Amazon MSK appropriés se produisent. Vous pouvez créer un mappage Amazon MSK à l'aide de la console. Créez un déclencheur Lambda, puis le mappage des sources d'événements est automatiquement configuré.

Pour créer un déclenceur Lambda (et un mappage des sources d'événements)

1. Accédez à la page de présentation de votre fonction Lambda.
2. Dans la section de présentation de la fonction, choisissez Ajouter un déclencheur en bas à gauche.
3. Dans le menu déroulant Sélectionner une source, sélectionnez Amazon MSK.
4. Ne configurez pas l'authentification.
5. Pour Cluster MSK, sélectionnez le nom de votre cluster.
6. Pour Taille de lot, saisissez 1. Cette étape facilite le test de cette fonctionnalité. Elle ne constitue pas une valeur idéale en production.
7. Pour Nom de la rubrique, indiquez le nom de votre rubrique Kafka.
8. Pour ID du groupe de consommateurs, indiquez l'ID de votre groupe de consommateurs Kafka.

Mise à jour de votre fonction Lambda pour lire vos données de streaming

Lambda fournit des informations sur les événements Kafka via le paramètre de méthode d'événement. Pour obtenir un exemple de structure d'un événement Amazon MSK, consultez the section called " Exemple d'évènement". Après avoir compris comment interpréter les événements Amazon MSK transférés par Lambda, vous pouvez modifier le code de votre fonction Lambda pour utiliser les informations qu'ils fournissent.

Fournissez le code suivant à votre fonction Lambda pour journaliser le contenu d'un événement Lambda Amazon MSK à des fins de test :

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon MSK avec Lambda en utilisant .NET.

```
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KafkaEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace MSKLambda;

public class Function
{


    /// <param name="input">The event for the Lambda function handler to
 process.</param>
    /// <param name="context">The ILambdaContext that provides methods for
 logging and describing the Lambda environment.</param>
    /// <returns></returns>
    public void FunctionHandler(KafkaEvent evnt, ILambdaContext context)
    {

        foreach (var record in evnt.Records)
        {
            Console.WriteLine("Key:" + record.Key);
            foreach (var eventRecord in record.Value)
            {
                var valueBytes = eventRecord.Value.ToArray();
                var valueText = Encoding.UTF8.GetString(valueBytes);
```

```
                Console.WriteLine("Message:" + valueText);
            }
        }
    }


}
```

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon MSK avec Lambda en utilisant Go.

```
package main

import (
 "encoding/base64"
 "fmt"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.KafkaEvent) {
 for key, records := range event.Records {
  fmt.Println("Key:", key)

  for _, record := range records {
   fmt.Println("Record:", record)

   decodedValue, _ := base64.StdEncoding.DecodeString(record.Value)
   message := string(decodedValue)
```

```
   fmt.Println("Message:", message)
  }
 }
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon MSK avec Lambda en utilisant Java.

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KafkaEvent;
import com.amazonaws.services.lambda.runtime.events.KafkaEvent.KafkaEventRecord;

import java.util.Base64;
import java.util.Map;

public class Example implements RequestHandler<KafkaEvent, Void> {

    @Override
    public Void handleRequest(KafkaEvent event, Context context) {
        for (Map.Entry<String, java.util.List<KafkaEventRecord>> entry :
 event.getRecords().entrySet()) {
            String key = entry.getKey();
            System.out.println("Key: " + key);

            for (KafkaEventRecord record : entry.getValue()) {
                System.out.println("Record: " + record);
```

```
                byte[] value = Base64.getDecoder().decode(record.getValue());
                String message = new String(value);
                System.out.println("Message: " + message);
            }
        }

        return null;
    }
}
```

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement Amazon MSK avec JavaScript Lambda à l'aide de.

```
exports.handler = async (event) => {
    // Iterate through keys
    for (let key in event.records) {
      console.log('Key: ', key)
      // Iterate through records
      event.records[key].map((record) => {
        console.log('Record: ', record)
        // Decode base64
        const msg = Buffer.from(record.value, 'base64').toString()
        console.log('Message:', msg)
      })
    }
}
```

Utilisation d'un événement Amazon MSK avec TypeScript Lambda à l'aide de.

```
import { MSKEvent, Context } from "aws-lambda";
```

```
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "msk-handler-sample",
});

export const handler = async (
  event: MSKEvent,
  context: Context
): Promise<void> => {
  for (const [topic, topicRecords] of Object.entries(event.records)) {
    logger.info(`Processing key: ${topic}`);

    // Process each record in the partition
    for (const record of topicRecords) {
      try {
        // Decode the message value from base64
        const decodedMessage = Buffer.from(record.value, 'base64').toString();

        logger.info({
          message: decodedMessage
        });
      }
      catch (error) {
        logger.error('Error processing event', { error });
        throw error;
      }
    };
  }
}
```

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon MSK avec Lambda en utilisant PHP.

```
<?php
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

// using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kafka\KafkaEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): void
    {
        $kafkaEvent = new KafkaEvent($event);
        $this->logger->info("Processing records");
        $records = $kafkaEvent->getRecords();
```

```
        foreach ($records as $record) {
            try {
                $key = $record->getKey();
                $this->logger->info("Key: $key");

                $values = $record->getValue();
                $this->logger->info(json_encode($values));

                foreach ($values as $value) {
                    $this->logger->info("Value: $value");
                }

            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon MSK avec Lambda en utilisant Python.

```
import base64

def lambda_handler(event, context):
    # Iterate through keys
```

```
    for key in event['records']:
        print('Key:', key)
        # Iterate through records
        for record in event['records'][key]:
            print('Record:', record)
            # Decode base64
            msg = base64.b64decode(record['value']).decode('utf-8')
            print('Message:', msg)
```

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon MSK avec Lambda en utilisant Ruby.

```
require 'base64'

def lambda_handler(event:, context:)
  # Iterate through keys
  event['records'].each do |key, records|
    puts "Key: #{key}"

    # Iterate through records
    records.each do |record|
      puts "Record: #{record}"

      # Decode base64
      msg = Base64.decode64(record['value'])
      puts "Message: #{msg}"
    end
  end
end
```

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon MSK avec Lambda en utilisant Rust.

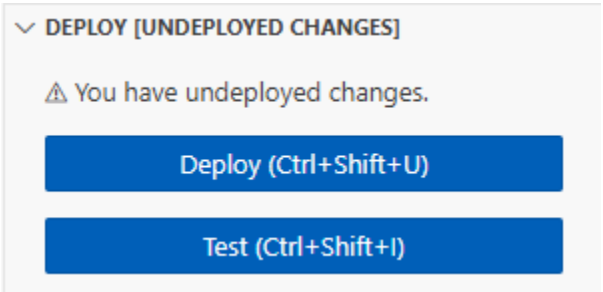
```
use aws_lambda_events::event::kafka::KafkaEvent;
use lambda_runtime::{run, service_fn, tracing, Error, LambdaEvent};
use base64::prelude::*;
use serde_json::{Value};
use tracing::{info};

/// Pre-Requisites:
/// 1. Install Cargo Lambda - see https://www.cargo-lambda.info/guide/getting-
started.html
/// 2. Add packages tracing, tracing-subscriber, serde_json, base64
///
/// This is the main body for the function.
/// Write your code inside it.
/// There are some code example in the following URLs:
/// - https://github.com/awslabs/aws-lambda-rust-runtime/tree/main/examples
/// - https://github.com/aws-samples/serverless-rust-demo/

async fn function_handler(event: LambdaEvent<KafkaEvent>) -> Result<Value, Error>
 {

    let payload = event.payload.records;

    for (_name, records) in payload.iter() {

        for record in records {

         let record_text = record.value.as_ref().ok_or("Value is None")?;
         info!("Record: {}", &record_text);

         // perform Base64 decoding
         let record_bytes = BASE64_STANDARD.decode(record_text)?;
```

```
            let message = std::str::from_utf8(&record_bytes)?;

            info!("Message: {}", message);
           }

    }

    Ok(().into())
}

#[tokio::main]
async fn main() -> Result<(), Error> {

    // required to enable CloudWatch error logging by the runtime
    tracing::init_default_subscriber();
    info!("Setup CW subscriber!");

    run(service_fn(function_handler)).await
}
```

Vous pouvez fournir le code de fonction à votre fonction Lambda à l'aide de la console.

Pour mettre à jour le code de fonction à l'aide de l'éditeur de code de la console

1. Ouvrez la page Fonctions de la console Lambda et choisissez votre fonction.
2. Sélectionnez l'onglet Code.
3. Dans le volet Source du code, sélectionnez votre fichier de code source et modifiez-le dans l'éditeur de code intégré.
4. Dans la section DÉPLOYER, choisissez Déployer pour mettre à jour le code de votre fonction :



Test de votre fonction Lambda pour vérifier qu'elle est connectée à votre rubrique Amazon MSK

Vous pouvez désormais vérifier si votre Lambda est invoqué par la source d'événements en consultant les journaux d'événements. CloudWatch

Pour vérifier si votre fonction Lambda est invoquée

1. Utilisez votre hôte administrateur Kafka pour générer des événements Kafka à l'aide de la CLI `kafka-console-producer`. Pour plus d'informations, consultez Write some events into the topic dans la documentation Kafka. Envoyez suffisamment d'événements pour remplir le lot défini en fonction de la taille du lot pour votre mappage des sources d'événements défini à l'étape précédente, sinon Lambda attendra d'autres informations pour procéder à l'invocation.
2. Si votre fonction s'exécute, Lambda écrit ce qui s'est passé à. CloudWatch Dans la console, accédez à la page des détails de votre fonction Lambda.
3. Sélectionnez l'onglet Configuration.
4. Dans la barre latérale, sélectionnez Outils de surveillance et d'exploitation.
5. Identifiez le groupe de CloudWatch journaux sous Configuration de la journalisation. Le groupe de journaux doit commencer par `/aws/lambda`. Choisissez le lien vers le groupe de journaux.
6. Dans la CloudWatch console, examinez les événements du journal pour les événements du journal que Lambda a envoyés au flux de journal. Identifiez s'il existe des événements de journaux contenant le message de votre événement Kafka, comme dans l'image suivante. Si tel est le cas, vous avez connecté avec succès une fonction Lambda à Amazon MSK à l'aide d'un mappage des sources d'événements Lambda.

Utilisation de Lambda avec Apache Kafka autogéré

Cette rubrique décrit comment utiliser Lambda avec un cluster Kafka autogéré. En AWS termes terminologiques, un cluster autogéré inclut les clusters Kafka non AWS hébergés. Par exemple, vous pouvez héberger votre cluster Kafka avec un fournisseur de cloud tel que Confluent Cloud ou Redpanda.

Ce chapitre explique comment utiliser un cluster Apache Kafka autogéré en tant que source d'événement pour votre fonction Lambda. Le processus général d'intégration d'Apache Kafka autogéré à Lambda comprend les étapes suivantes :

1. Configuration du cluster et du réseau : configurez d'abord votre cluster Apache Kafka autogéré avec la configuration réseau appropriée pour permettre à Lambda d'accéder à votre cluster.
2. Configuration du mappage des sources d'événements : créez ensuite la ressource de mappage des sources d'événements dont Lambda a besoin pour connecter de façon sécurisée votre cluster Apache Kafka à votre fonction.
3. Configuration de la fonction et des autorisations : enfin, assurez-vous que votre fonction est correctement configurée et qu'elle dispose des autorisations nécessaires dans son rôle d'exécution.

Apache Kafka en tant que source d'événement fonctionne de la même manière qu'Amazon Simple Queue Service (Amazon SQS) ou Amazon Kinesis. Lambda interroge en interne les nouveaux messages de la source d'événement, puis invoque de manière synchrone la fonction Lambda cible. Lambda lit les messages par lot et les fournit à votre fonction en tant que charge utile d'événement. La taille maximale du lot est configurable (la valeur par défaut est de 100 messages). Pour de plus amples informations, veuillez consulter the section called "Comportement de traitement par lots".

Pour optimiser le débit de votre mappage des sources d'événements Apache Kafka autogéré, configurez le mode alloué. En mode alloué, vous pouvez définir le nombre minimal et maximal de sondeurs d'événements alloués à votre mappage des sources d'événements. Cela peut améliorer la capacité de votre mappage des sources d'événements à gérer les pics de messages inattendus. Pour de plus amples informations, veuillez consulter Mode alloué.

> **Warning**
>
> Les mappages des sources d'événements Lambda traitent chaque événement au moins une fois, et le traitement des enregistrements peut être dupliqué. Pour éviter les problèmes potentiels liés à des événements dupliqués, nous vous recommandons vivement de rendre

votre code de fonction idempotent. Pour en savoir plus, consultez Comment rendre ma fonction Lambda idempotente dans le Knowledge Center. AWS

Pour les sources d'événements basées sur Kafka, Lambda prend en charge les paramètres de contrôle du traitement par lots, tels que les fenêtres de traitement par lots et la taille des lots. Pour de plus amples informations, veuillez consulter Comportement de traitement par lots.

Pour un exemple d'utilisation de Kafka autogéré comme source d'événements, consultez la section Utilisation d'Apache Kafka auto-hébergée comme source d'événements AWS Lambda sur le blog Compute. AWS

Rubriques

- Exemple d'évènement
- Configuration de votre cluster Apache Kafka autogéré, de votre VPC et de votre réseau Lambda
- Configuration des autorisations de rôle d'exécution Lambda
- Configuration de sources d'événements Apache Kafka autogérées pour Lambda

Exemple d'évènement

Lambda envoie le lot de messages dans le paramètre d'événement quand il invoque votre fonction Lambda. La charge utile d'un événement contient un tableau de messages. Chaque élément de tableau contient des détails de la rubrique Kafka et l'identifiant de partition Kafka, ainsi qu'un horodatage et un message codé en base 64.

```
{
   "eventSource": "SelfManagedKafka",
   "bootstrapServers":"b-2.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092",
   "records":{
      "mytopic-0":[
         {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "key":"abcDEFghiJKLmnoPQRstuVWXyz1234==",
```

```
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[
               {
                  "headerKey":[
                     104,
                     101,
                     97,
                     100,
                     101,
                     114,
                     86,
                     97,
                     108,
                     117,
                     101
                  ]
               }
            ]
         }
      ]
   }
}
```

Configuration de votre cluster Apache Kafka autogéré, de votre VPC et de votre réseau Lambda

Pour connecter votre fonction Lambda à votre cluster Apache Kafka autogéré, vous devez configurer correctement votre cluster et le réseau dans lequel il réside. Cette page décrit comment configurer votre cluster et votre réseau. Si votre cluster et votre réseau sont déjà correctement configurés, consultez the section called “Configurer une source d’événements” pour configurer le mappage des sources d’événements.

Rubriques

- Configuration de cluster Apache Kafka autogéré
- Configurer la sécurité réseau

Configuration de cluster Apache Kafka autogéré

Vous pouvez héberger votre cluster Apache Kafka autogéré auprès de fournisseurs de cloud comme Confluent Cloud ou Redpanda, ou l’exécuter sur votre propre infrastructure. Assurez-vous que votre

cluster est correctement configuré et accessible depuis le réseau auquel votre mappage des sources d'événements Lambda sera connecté.

Configurer la sécurité réseau

Pour donner à Lambda un accès complet à Apache Kafka autogéré via votre mappage des sources d'événements, soit votre cluster doit utiliser un point de terminaison public (adresse IP publique), soit vous devez fournir un accès au VPC Amazon dans lequel vous avez créé le cluster.

Lorsque vous utilisez Apache Kafka autogéré avec Lambda, créez des points de terminaison de VPC AWS PrivateLink qui permettent à votre fonction d'accéder aux ressources de votre Amazon VPC.

> **Note**
>
> Les points de terminaison de VPC AWS PrivateLink sont requis pour les fonctions avec des mappages de sources d'événements qui utilisent le mode par défaut (à la demande) pour les interrogateurs d'événements. Si le mappage de votre source d'événements utilise le mode provisionné, vous n'avez pas besoin de configurer les points de terminaison de VPC AWS PrivateLink.

Créez un point de terminaison pour accéder aux ressources suivantes :

- Lambda — Créez un point de terminaison pour le principal de service Lambda.
- AWS STS — Créez un point de terminaison pour l'AWS STS afin qu'un principal de service assume un rôle en votre nom.
- Secrets Manager : si votre cluster utilise Secrets Manager pour stocker les informations d'identification, créez un point de terminaison pour Secrets Manager.

Vous pouvez également configurer une passerelle NAT sur chaque sous-réseau public d'Amazon VPC. Pour de plus amples informations, consultez the section called "Accès Internet pour les fonctions VPC".

Lorsque vous créez un mappage des sources d'événements pour Apache Kafka autogéré, Lambda vérifie si des interfaces réseau Elastic (ENI) sont déjà présentes pour les sous-réseaux et les groupes de sécurité configurés pour votre VPC Amazon. Si Lambda trouve des ENI existantes, il tente de les réutiliser. Sinon, Lambda crée de nouvelles ENI pour se connecter à la source d'événement et invoquer votre fonction.

> **Note**
>
> Les fonctions Lambda sont toujours exécutées dans des VPC appartenant au service Lambda. La configuration VPC de votre fonction n'affecte pas le mappage des sources d'événements. Seule la configuration réseau des sources d'événements détermine la manière dont Lambda se connecte à votre source d'événements.

Configurez les groupes de sécurité pour le VPC Amazon contenant votre cluster. Par défaut, Apache Kafka autogéré utilise les ports suivants : `9092`.

- Règles entrantes – Autorisent tout le trafic sur le port de l'agent par défaut pour le groupe de sécurité associé à votre source d'événement. Vous pouvez également utiliser une règle de groupe de sécurité avec référence circulaire pour autoriser l'accès à partir d'instances appartenant au même groupe de sécurité.
- Règles sortantes : autorisent tout le trafic sur le port `443` pour des destinations externes si votre fonction doit communiquer avec des services AWS. Vous pouvez également utiliser une règle de groupe de sécurité avec référence circulaire pour limiter l'accès à l'agent si vous n'avez pas besoin de communiquer avec d'autres services AWS.
- Règles entrantes relatives au point de terminaison Amazon VPC – Si vous utilisez un point de terminaison Amazon VPC, le groupe de sécurité associé à votre point de terminaison Amazon VPC doit autoriser le trafic entrant sur le port `443` en provenance du groupe de sécurité du cluster.

Si votre cluster utilise l'authentification, vous pouvez également restreindre la politique de point de terminaison pour le point de terminaison Secrets Manager. Pour appeler l'API Secrets Manager, Lambda utilise votre rôle de fonction, et non le principal de service Lambda.

Example Politique de point de terminaison de VPC — Point de terminaison Secrets Manager

```
{
      "Statement": [
          {
              "Action": "secretsmanager:GetSecretValue",
              "Effect": "Allow",
              "Principal": {
                  "AWS": [
                      "arn:aws::iam::123456789012:role/my-role"
                  ]
```

```
            },
            "Resource": "arn:aws::secretsmanager:us-west-2:123456789012:secret:my-
secret"
        }
    ]
}
```

Lorsque vous utilisez des points de terminaison Amazon VPC, AWS achemine vos appels d'API pour invoquer votre fonction à l'aide de l'interface réseau Elastic (ENI) du point de terminaison. Le principal de service Lambda doit appeler `lambda:InvokeFunction` pour tous les rôles et fonctions utilisant ces ENI.

Par défaut, les points de terminaison Amazon VPC disposent de politiques IAM ouvertes qui permettent un accès étendu aux ressources. La meilleure pratique consiste à restreindre ces politiques pour effectuer les actions nécessaires à l'aide de ce point de terminaison. Pour garantir que votre mappage des sources d'événements est en mesure d'invoquer votre fonction Lambda, la politique de point de terminaison de VPC doit autoriser le principal de service Lambda à appeler `sts:AssumeRole` et `lambda:InvokeFunction`. Le fait de restreindre vos politiques de point de terminaison de VPC pour autoriser uniquement les appels d'API provenant de votre organisation empêche le mappage des sources d'événements de fonctionner correctement. C'est pourquoi `"Resource": "*"` est requis dans ces politiques.

Les exemples de politiques de point de terminaison de VPC suivants montrent comment accorder l'accès requis au principal de service Lambda pour les points de terminaison AWS STS et Lambda.

Example Politique de point de terminaison de VPC — Point de terminaison AWS STS

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Example Politique de point de terminaison de VPC — Point de terminaison Lambda

```
{
      "Statement": [
          {
              "Action": "lambda:InvokeFunction",
              "Effect": "Allow",
              "Principal": {
                  "Service": [
                      "lambda.amazonaws.com"
                  ]
              },
              "Resource": "*"
          }
      ]
  }
```

Configuration des autorisations de rôle d'exécution Lambda

En plus d'accéder au cluster Amazon Kafka autogéré, votre fonction Lambda a besoin d'autorisations pour effectuer diverses actions d'API. Ajoutez ces autorisations au rôle d'exécution de votre fonction. Si vos utilisateurs ont besoin d'accéder à des actions d'API, ajoutez les autorisations requises à la politique d'identité de l'utilisateur ou du rôle Gestion des identités et des accès AWS (IAM).

Rubriques

- Autorisations de fonction Lambda requises
- Autorisations de fonction Lambda facultatives
- Ajout d'autorisations à votre rôle d'exécution
- Octroi d'accès à des utilisateurs avec une politique IAM

Autorisations de fonction Lambda requises

Pour créer et stocker des journaux dans un groupe de CloudWatch journaux dans Amazon Logs, votre fonction Lambda doit disposer des autorisations suivantes dans son rôle d'exécution :

- journaux : CreateLogGroup
- journaux : CreateLogStream
- journaux : PutLogEvents

Autorisations de fonction Lambda facultatives

Votre fonction Lambda peut également nécessiter ces autorisations pour :

- Décrivez votre secret Secrets Manager.
- Accédez à votre AWS Key Management Service (AWS KMS) clé gérée par le client.
- Accédez à votre Amazon VPC.
- Envoyez les enregistrements des invocations ayant échoué vers une destination.

Secrets Manager et AWS KMS autorisations

Selon le type de contrôle d'accès que vous configurez pour vos courtiers Kafka, votre fonction Lambda peut avoir besoin d'une autorisation pour accéder à votre secret Secrets Manager ou pour déchiffrer AWS KMS votre clé gérée par le client. Afin d'accéder à ces ressources, le rôle d'exécution de votre fonction doit disposer des autorisations suivantes :

- responsable des secrets : GetSecretValue
- kms:Decrypt

Autorisations VPC

Si seuls des utilisateurs au sein d'un VPC peuvent accéder à votre cluster Apache Kafka autogéré, votre fonction Lambda doit être autorisée à accéder à vos ressources Amazon VPC. Ces ressources incluent les sous-réseaux, groupes de sécurité et interfaces réseau de votre VPC. Afin d'accéder à ces ressources, le rôle d'exécution de votre fonction doit disposer des autorisations suivantes :

- EC2 : CreateNetworkInterface
- EC2 : DescribeNetworkInterfaces
- EC2 : DescribeVpcs
- EC2 : DeleteNetworkInterface
- EC2 : DescribeSubnets
- EC2 : DescribeSecurityGroups

Ajout d’autorisations à votre rôle d’exécution

Pour accéder aux autres AWS services utilisés par votre cluster Apache Kafka autogéré, Lambda utilise les politiques d'autorisation que vous définissez dans le rôle d'exécution de votre fonction Lambda.

Par défaut, Lambda n’est pas autorisé à exécuter les actions obligatoires ou facultatives pour un cluster Apache Kafka autogéré. Vous devez créer et définir ces actions dans une stratégie d’approbation IAM pour votre rôle d’exécution. Cet exemple montre comment créer une stratégie autorisant Lambda à accéder à vos ressources Amazon VPC.

JSON

```
{
        "Version":"2012-10-17",
        "Statement":[
           {
              "Effect":"Allow",
              "Action":[
                 "ec2:CreateNetworkInterface",
                 "ec2:DescribeNetworkInterfaces",
                 "ec2:DescribeVpcs",
                 "ec2:DeleteNetworkInterface",
                 "ec2:DescribeSubnets",
                 "ec2:DescribeSecurityGroups"
              ],
              "Resource":"*"
           }
        ]
    }
```

Octroi d’accès à des utilisateurs avec une politique IAM

Par défaut, les utilisateurs et les rôles n’ont pas l’autorisation d’effectuer des opérations d’API de source d’événement. Pour accorder l’accès aux utilisateurs de votre organisation ou de votre compte, vous pouvez ajouter ou mettre à jour une stratégie basée sur l’identité. Pour plus d'informations, consultez la section Contrôle de l'accès aux AWS ressources à l'aide de politiques dans le Guide de l'utilisateur IAM.

Pour résoudre les erreurs d'authentification et d'autorisation, consultez the section called "Résolution des problèmes".

Configuration de sources d'événements Apache Kafka autogérées pour Lambda

Pour utiliser un cluster Apache Kafka autogéré en tant que source d'événements pour votre fonction Lambda, vous devez créer un mappage des sources d'événements qui connecte les deux ressources. Cette page explique comment créer un mappage des sources d'événements pour Apache Kafka autogéré.

Cette page suppose que vous avez déjà correctement configuré votre cluster Kafka et le réseau dans lequel il réside. Si vous devez configurer votre cluster ou votre réseau, consultez the section called "Configuration du cluster et du réseau".

Rubriques

- Utilisation d'un cluster Apache Kafka autogéré comme source d'événement
- Configuration des méthodes d'authentification de cluster dans Lambda
- Création d'un mappage des sources d'événements Lambda pour une source d'événements Apache Kafka autogéré
- Tous les paramètres de configuration des sources d'événements Apache Kafka autogéré dans Lambda

Utilisation d'un cluster Apache Kafka autogéré comme source d'événement

Lorsque vous ajoutez votre cluster Apache Kafka ou Amazon MSK comme déclencheur pour votre fonction Lambda, le cluster est utilisé comme source d'événement.

Lambda lit les données d'événements des rubriques Kafka que vous spécifiez `Topics` dans une CreateEventSourceMappingdemande, en fonction de la position de départ que vous spécifiez. Lorsque le traitement a réussi, votre rubrique Kafka est validée dans votre cluster Kafka.

Lambda lit les messages séquentiellement pour chaque partition de rubrique Kafka. Une seule charge utile Lambda peut contenir des messages provenant de plusieurs partitions. Lorsque d'autres enregistrements sont disponibles, Lambda continue de traiter les enregistrements par lots, en fonction de la BatchSize valeur que vous spécifiez dans une CreateEventSourceMappingdemande, jusqu'à ce que votre fonction aborde le sujet.

Après avoir traité chaque lot, Lambda valide les décalages des messages dans celui-ci. Si votre fonction renvoie une erreur pour l'un des messages d'un lot, Lambda réessaie le lot de messages

complet jusqu'à ce que le traitement réussisse ou que les messages expirent. Vous pouvez envoyer les enregistrements qui échouent à toutes les tentatives vers une destination en cas de panne pour un traitement ultérieur.

> **Note**
>
> Alors que les fonctions Lambda ont généralement un délai d'expiration maximal de 15 minutes, les mappages des sources d'événement pour Amazon MSK, Apache Kafka autogéré, Amazon DocumentDB et Amazon MQ pour ActiveMQ et RabbitMQ ne prennent en charge que les fonctions dont le délai d'expiration maximal est de 14 minutes.

Configuration des méthodes d'authentification de cluster dans Lambda

Lambda prend en charge plusieurs méthodes d'authentification auprès de votre cluster Apache Kafka autogéré. Veillez à configurer le cluster Kafka pour utiliser l'une des méthodes d'authentification prises en charge suivantes : Pour de plus amples informations sur la sécurité, veuillez consulter la section Sécurité de la documentation Kafka.

Authentification SASL/SCRAM

Lambda prend en charge l'authentification simple et le mécanisme d'authentification par réponse aux Layer/Salted défis de sécurité (SASL/SCRAM) avec le chiffrement TLS (Transport Layer Security) (). `SASL_SSL` Lambda envoie les informations d'identification chiffrées pour s'authentifier auprès du cluster. Lambda n'est pas compatible SASL/SCRAM avec plaintext (). `SASL_PLAINTEXT` Pour plus d'informations sur SASL/SCRAM l'authentification, consultez la RFC 5802.

Lambda prend également en charge SASL/PLAIN l'authentification. Comme ce mécanisme utilise des informations d'identification en texte clair, la connexion au serveur doit utiliser le chiffrement TLS pour garantir la protection des informations d'identification.

Pour l'authentification SASL, vous devez stocker les informations d'identification en tant que secret dans AWS Secrets Manager. Pour plus d'informations sur l'utilisation de Secrets Manager, consultez la section Créer un AWS Secrets Manager secret dans le Guide de AWS Secrets Manager l'utilisateur.

> **Important**
>
> Pour utiliser Secrets Manager pour l'authentification, les secrets doivent être stockés dans la même AWS région que votre fonction Lambda.

Authentification TLS mutuelle

Mutual TLS (mTLS) fournit une authentification bidirectionnelle entre le client et le serveur. Le client envoie un certificat au serveur pour que le serveur vérifie le client, et le serveur envoie un certificat au client pour que le client vérifie le serveur.

Dans Apache Kafka autogéré, Lambda agit en tant que client. Vous configurez un certificat client (en tant que secret dans Secrets Manager) pour authentifier Lambda auprès des courtiers Kafka. Le certificat client doit être signé par une autorité de certification dans le magasin d'approbations du serveur.

Le cluster Kafka envoie un certificat de serveur à Lambda pour authentifier les courtiers auprès de Lambda. Le certificat de serveur peut être un certificat d'autorité de certification public ou privéCA/ self-signed certificate. The public CA certificate must be signed by a certificate authority (CA) that's in the Lambda trust store. For a private CA/self, vous configurez le certificat d'autorité de certification racine du serveur (en tant que secret dans Secrets Manager). Lambda utilise le certificat racine pour vérifier les courtiers Kafka.

Pour de plus amples informations sur mTLS, veuillez consulter Introduction d'une authentification TLS mutuelle pour Amazon MSK en tant que source d'événement.

Configuration du secret du certificat client

Le secret CLIENT_CERTIFICATE_TLS_AUTH nécessite un champ de certificat et un champ de clé privée. Pour une clé privée chiffrée, le secret nécessite un mot de passe de clé privée. Le certificat et la clé privée doivent être au format PEM.

> **Note**
>
> Lambda prend en charge les algorithmes de chiffrement par clé privée PBES1(mais pas PBES2).

Le champ de certificat doit contenir une liste de certificats, commençant par le certificat client, suivi de tous les certificats intermédiaires et se terminant par le certificat racine. Chaque certificat doit commencer sur une nouvelle ligne avec la structure suivante :

```
-----BEGIN CERTIFICATE-----
            <certificate contents>
-----END CERTIFICATE-----
```

Secrets Manager prend en charge les secrets jusqu'à 65 536 octets, ce qui offre suffisamment d'espace pour de longues chaînes de certificats.

La clé privée doit être au format PKCS #8, avec la structure suivante :

```
-----BEGIN PRIVATE KEY-----
             <private key contents>
-----END PRIVATE KEY-----
```

Pour une clé privée chiffrée, utilisez la structure suivante :

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
              <private key contents>
-----END ENCRYPTED PRIVATE KEY-----
```

L'exemple suivant affiche le contenu d'un secret pour l'authentification mTLS à l'aide d'une clé privée chiffrée. Pour une clé privée chiffrée, incluez le mot de passe de clé privée dans le secret.

```
{"privateKeyPassword":"testpassword",
"certificate":"-----BEGIN CERTIFICATE-----
MIIE5DCCAsygAwIBAgIRAPJdwaFaNRrytHBto0j5BA0wDQYJKoZIhvcNAQELBQAw
...
j0Lh4/+1HfgyE2KlmII36dg4IMzNjAFEBZiCRoPimO40s1cRqtFHXoal0QQbIlxk
cmUuiAii9R0=
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIFgjCCA2qgAwIBAgIQdjNZd6uFf9hbNC5RdfmHrzANBgkqhkiG9w0BAQsFADBb
...
rQoiowbbk5wXCheYSANQIfTZ6weQTgiCHCCbuuMKNVS95FkXm0vqVD/YpXKwA/no
c8PH3PSoAaRwMMgOSA2ALJvbRz8mpg==
-----END CERTIFICATE-----",
"privateKey":"-----BEGIN ENCRYPTED PRIVATE KEY-----
MIIFKzBVBgkqhkiG9w0BBQ0wSDAnBgkqhkiG9w0BBQwwGgQUiAFcK5hT/X7Kjmgp
```

```
...
QrSekqF+kWzmB6nAfSzgO9IaoAaytLvNgGTckWeUkWn/V0Ck+LdGUXzAC4RxZnoQ
zp2mwJn2NYB7AZ7+imp0azDZb+8YG2aUCiyqb6PnnA==
-----END ENCRYPTED PRIVATE KEY-----"
}
```

Configuration du secret du certificat d'autorité de certification racine du serveur

Vous créez ce secret si vos courtiers Kafka utilisent le chiffrement TLS avec des certificats signés par une autorité de certification privée. Vous pouvez utiliser le chiffrement TLS pour l'authentification VPC ou SASL/SCRAM, SASL/PLAIN mTLS.

Le secret du certificat d'autorité de certification racine du serveur nécessite un champ contenant le certificat d'autorité de certification racine du courtier Kafka au format PEM. La structure du secret est présentée dans l'exemple suivant.

```
{"certificate":"-----BEGIN CERTIFICATE-----
MIID7zCCAtegAwIBAgIBADANBgkqhkiG9w0BAQsFADCBmDELMAkGA1UEBhMCVVMx
EDAOBgNVBAgTB0FyaXpvbmExEzARBgNVBAcTClNjb3R0c2RhbGUxJTAjBgNVBAoT
HFN0YXJmaWVsZCBUZWNobm9sb2dpZXMsIEluYy4xOzA5BgNVBAMTMlN0YXJmaWVs
ZCBTZXJ2aWNlcyBSb290IENlcnRpZmljYXRlIEF1dG...
-----END CERTIFICATE-----"
}
```

Création d'un mappage des sources d'événements Lambda pour une source d'événements Apache Kafka autogéré

Pour créer un mappage des sources d'événements, vous pouvez utiliser la console Lambda, AWS Command Line Interface (CLI) ou un SDK AWS.

Les étapes de console suivantes ajoutent un cluster Apache Kafka autogéré en tant que déclencheur pour votre fonction Lambda. En arrière-plan, cela crée une ressource de mappage des sources d'événements.

Conditions préalables

- Cluster Apache Kafka autogéré. Lambda prend en charge Apache Kafka versions 0.10.1.0 et ultérieures.
- Rôle d'exécution autorisé à accéder aux AWS ressources utilisées par votre cluster Kafka autogéré.

Ajout d’un cluster Kafka autogéré (console)

Pour ajouter votre cluster Apache Kafka autogéré et une rubrique Kafka en tant que déclencheur pour votre fonction Lambda, procédez comme suit.

Pour ajouter un déclencheur Apache Kafka à votre fonction Lambda (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom de votre fonction Lambda.
3. Sous Function overview (Vue d’ensemble de la fonction), choisissez Add trigger (Ajouter un déclencheur).
4. Sous Trigger configuration (Configuration du déclencheur), procédez comme suit :
    a. Choisissez le type de déclencheur Apache Kafka.
    b. Pour Bootstrap servers (Serveurs d’amorçage), entrez l’adresse de paire hôte et port d’un agent Kafka dans votre cluster, puis choisissez Add (Ajouter). Répétez l’opération pour chaque agent Kafka dans le cluster.
    c. Pour Topic name (Nom de rubrique), entrez le nom de la rubrique Kafka utilisée pour stocker les registres dans le cluster.
    d. Si vous configurez le mode provisionné, entrez une valeur pour Minimum Event Sollers, une valeur pour Maximum Event Sollers et une valeur facultative pour PollerGroupName spécifier le regroupement de plusieurs ESMs au sein d'un même VPC source d'événements.
    e. (Facultatif) Pour Batch size (Taille de lot), entrez le nombre maximal de registres à recevoir dans un même lot.
    f. Pour Batch window, veuillez saisir l’intervalle de temps maximal en secondes nécessaire à Lambda pour collecter des enregistrements avant d’invoquer la fonction.
    g. (Facultatif) Pour l’identifiant de groupe de consommateurs, entrez l’identifiant d’un groupe de consommateurs Kafka à rejoindre.
    h. (Facultatif) Pour Position de départ, choisissez Dernier pour commencer à lire le flux à partir du dernier enregistrement, Supprimer l’horizon pour commencer au premier enregistrement disponible ou À l’horodatage pour spécifier un horodatage à partir duquel commencer la lecture.
    i. (Facultatif) Pour VPC, choisissez l’Amazon VPC pour votre cluster Kafka. Ensuite, choisissez le VPC subnets (Sous-réseaux VPC) et les VPC security groups (Groupes de sécurité VPC).

Ce paramètre est requis si seuls des utilisateurs au sein de votre VPC accèdent à vos courtiers.

j. (Facultatif) Pour Authentication (Authentification), choisissez Add (Ajouter), puis procédez comme suit :

   i. Choisissez le protocole d’accès ou d’authentification des courtiers Kafka dans votre cluster.

      - Si votre broker Kafka utilise l' SASL/PLAIN authentification, choisissez BASIC_AUTH.
      - Si votre courtier utilise SASL/SCRAM l'authentification, choisissez l'un des SASL_SCRAMprotocoles.
      - Si vous configurez l’authentification mTLS, choisissez le protocole CLIENT_CERTIFICATE_TLS_AUTH.

   ii. Pour l' SASL/SCRAM authentification mTLS, choisissez la clé secrète de Secrets Manager qui contient les informations d'identification de votre cluster Kafka.

k. (Facultatif) Pour Encryption (Chiffrement), choisissez le secret Secrets Manager contenant le certificat d’autorité de certification racine que vos courtiers Kafka utilisent pour le chiffrement TLS, si vos courtiers Kafka utilisent des certificats signés par une autorité de certification privée.

   Ce paramètre s'applique au chiffrement TLS pour SASL/SCRAM ou SASL/PLAIN, ainsi qu'à l'authentification MTLS.

l. Pour créer le déclencheur dans un état désactivé pour le test (recommandé), désactivez Enable trigger (Activer le déclencheur). Ou, pour activer le déclencheur immédiatement, sélectionnezActiver un déclencheur.

5. Pour créer le déclencheur, choisissez Add (Ajouter).

Ajout d’un cluster Kafka autogéré (AWS CLI)

Utilisez les exemples de AWS CLI commandes suivants pour créer et afficher un déclencheur Apache Kafka autogéré pour votre fonction Lambda.

Utilisation de SASL/SCRAM

Si les utilisateurs de Kafka accèdent à vos courtiers Kafka via Internet, spécifiez le secret du Gestionnaire de Secrets que vous avez créé pour SASL/SCRAM l'authentification. L'exemple suivant utilise la create-event-source-mapping AWS CLI commande pour mapper une fonction Lambda nommée `my-kafka-function` à une rubrique Kafka nommée. `AWSKafkaTopic`

```
aws lambda create-event-source-mapping \
  --topics AWSKafkaTopic \
  --source-access-configuration Type=SASL_SCRAM_512_AUTH,URI=arn:aws:secretsmanager:us-
east-1:111122223333:secret:MyBrokerSecretName \
  --function-name arn:aws:lambda:us-east-1:111122223333:function:my-kafka-function \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":
["abc3.xyz.com:9092", "abc2.xyz.com:9092"]}}'
```

Utilisation d’un VPC

Si seuls des utilisateurs de Kafka au sein de votre VPC accèdent à vos agents Kafka, vous devez spécifier votre VPC, vos sous-réseaux et votre groupe de sécurité de VPC. L'exemple suivant utilise la create-event-source-mapping AWS CLI commande pour mapper une fonction Lambda nommée `my-kafka-function` à une rubrique Kafka nommée. `AWSKafkaTopic`

```
aws lambda create-event-source-mapping \
  --topics AWSKafkaTopic \
  --source-access-configuration '[{"Type": "VPC_SUBNET", "URI":
 "subnet:subnet-0011001100"}, {"Type": "VPC_SUBNET", "URI":
 "subnet:subnet-0022002200"}, {"Type": "VPC_SECURITY_GROUP", "URI":
 "security_group:sg-0123456789"}]' \
  --function-name arn:aws:lambda:us-east-1:111122223333:function:my-kafka-function \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":
["abc3.xyz.com:9092", "abc2.xyz.com:9092"]}}'
```

Affichage de l'état à l'aide du AWS CLI

L'exemple suivant utilise la get-event-source-mapping AWS CLI commande pour décrire l'état du mappage des sources d'événements que vous avez créé.

```
aws lambda get-event-source-mapping
            --uuid dh38738e-992b-343a-1077-3478934hjkfd7
```

Tous les paramètres de configuration des sources d'événements Apache Kafka autogéré dans Lambda

Tous les types de sources d'événements Lambda partagent les mêmes opérations CreateEventSourceMappinget les mêmes opérations d'UpdateEventSourceMappingAPI. Cependant, seuls certains paramètres s'appliquent à Apache Kafka autogéré, comme indiqué dans le tableau suivant.

Paramètre	Obligatoire	Par défaut	Remarques
BatchSize	N	100	Maximum : 10 000.
DestinationConfig	N	N/A	the section called "Conserver les appels ayant échoué"
Activé	N	True	
FilterCriteria	N	N/A	Contrôle des événements envoyés par Lambda à votre fonction
FunctionName	Y	N/A	
KMSKeyArn	N	N/A	the section called "Chiffrement des critères de filtre"
MaximumBatchingWindowInSeconds	N	500 ms	Comportement de traitement par lots
ProvisionedPollersConfig	N	`MinimumPollers` : la valeur par défaut, si elle n'est pas spécifiée, est de 1 `MaximumPollers` : la valeur par défaut,	the section called "Mode alloué"

Paramètre	Obligatoire	Par défaut	Remarques
		si elle n’est pas spécifiée, est de 200 `PollerGroupName` : N/A	
SelfManagedEventSource	Y	N/A	Liste des agents Kafka. Peut définir uniquement sur Create (Créer)
SelfManagedKafkaEventSourceConfig	N	Contient le ConsumerGroupId champ qui prend par défaut une valeur unique.	Peut définir uniquement sur Create (Créer)
SourceAccessConfigurations	N	Pas d’informations d’identification	Informations sur le VPC ou informations d’authentification pour le cluster Pour SASL_PLAIN, défini sur BASIC_AUTH
StartingPosition	Y	N/A	AT_TIMESTAMP, TRIM_HORIZON ou DERNIER Peut définir uniquement sur Create (Créer)

Paramètre	Obligatoire	Par défaut	Remarques
StartingPositionTimestamp	N	N/A	Obligatoire s'il StartingPosition est défini sur AT_TIMESTAMP
Étiquettes	N	N/A	the section called “Balises de mappage des sources d’événements”
Rubriques	Y	N/A	Nom du sujet Peut définir uniquement sur Create (Créer)

> **Note**
>
> Lorsque vous spécifiez un`PollerGroupName`, plusieurs ESMs au sein d'un même Amazon VPC peuvent partager la capacité de l'Event Poller Unit (EPU). Vous pouvez utiliser cette option pour optimiser les coûts du mode provisionné pour votre ESMs. Exigences relatives au regroupement ESM :
>
> - ESMs doit se trouver dans le même Amazon VPC
> - Maximum de 100 ESMs par groupe de sondeurs
> - Le nombre maximum de sondeurs agrégés pour l'ensemble ESMs d'un groupe ne peut pas dépasser 2 000
>
> Vous pouvez le mettre `PollerGroupName` à jour pour déplacer un ESM vers un autre groupe, ou supprimer un ESM d'un groupe en le définissant `PollerGroupName` sur une chaîne vide (« »).

Modes de mise à l'échelle des sondeurs d'événements Apache Kafka dans Lambda

Vous pouvez choisir entre deux modes de mise à l'échelle des sondeurs d'événements pour Amazon MSK et les mappages des sources d'événements Apache Kafka autogéré :

- Mode à la demande (par défaut)
- Mode alloué

Mode à la demande (par défaut)

Lorsque vous créez initialement une source d'événement Kafka, Lambda alloue un nombre de sondeurs d'événements par défaut pour traiter toutes les partitions de la rubrique Kafka. Lambda augmente ou diminue automatiquement le nombre de sondeurs d'événements, en fonction de la charge de messages.

Toutes les minutes, Lambda évalue le décalage entre toutes les partitions dans la rubrique. Si le décalage est trop élevé, la partition reçoit des messages plus rapidement que Lambda ne peut les traiter. Si nécessaire, Lambda ajoute ou supprime des sondeurs d'événements dans la rubrique. Cette mise à l'échelle automatique consistant à ajouter ou à supprimer des sondeurs d'événements a lieu dans les trois minutes suivant l'évaluation.

Si votre fonction Lambda cible est limitée, Lambda réduit le nombre de sondeurs d'événements. Cette action réduit la charge de travail de la fonction en diminuant le nombre de messages que les sondeurs d'événements peuvent échanger avec la fonction.

Mode alloué

Pour les charges de travail où vous devez optimiser le débit de votre mappage des sources d'événements, vous pouvez utiliser le mode provisionné. En mode provisionné, vous définissez des limites minimales et maximales pour le nombre de sondeurs d'événements alloués. Ces sondeurs d'événements alloués sont dédiés à votre mappage des sources d'événements et peuvent gérer les pics de messages inattendus grâce à une mise à l'échelle automatique réactive. Nous vous recommandons d'utiliser le mode provisionné pour les charges de travail Kafka soumises à des exigences de performance strictes.

Dans Lambda, un sondeur d'événements est une unité de calcul dont les capacités de débit varient selon le type de source d'événements. Pour Amazon MSK et Apache Kafka autogéré, chaque

sondeur d'événements peut gérer jusqu'à 5 % MB/sec du débit ou jusqu'à 5 appels simultanés. Par exemple, si la source de votre événement produit une charge utile moyenne de 1 Mo et que la durée moyenne de votre fonction est de 1 seconde, un seul sondeur d'événements Kafka peut prendre en charge 5 MB/sec débits et 5 appels Lambda simultanés (en supposant qu'il n'y ait aucune transformation de charge utile). Pour Amazon SQS, chaque sondeur d'événements peut gérer jusqu'à 1 % MB/sec du débit ou jusqu'à 10 appels simultanés. L'utilisation du mode provisionné entraîne des coûts supplémentaires en fonction de l'utilisation de votre sondeur d'événements. Pour plus d'informations sur la tarification, consultez Tarification AWS Lambda.

> **Note**
>
> Lorsque vous utilisez le mode provisionné, il n'est pas nécessaire de créer des points de terminaison AWS PrivateLink VPC ni d'accorder les autorisations associées dans le cadre de la configuration de votre réseau.

En mode provisionné, la plage de valeurs acceptées pour le nombre minimal de sondeurs d'événements (`MinimumPollers`) est comprise entre 1 et 200 inclus. La plage de valeurs acceptées pour le nombre maximal de sondeurs d'événements (`MaximumPollers`) est comprise entre 1 et 2 000 inclus. `MaximumPollers` doit être supérieur ou égal à `MinimumPollers`. En outre, pour maintenir un traitement ordonné au sein des partitions, Lambda limite le nombre de `MaximumPollers` au nombre de partitions dans la rubrique.

Pour plus de détails sur le choix des valeurs appropriées pour le nombre minimal et maximal de sondeurs d'événements, consultez Bonnes pratiques.

Vous pouvez configurer le mode provisionné pour le mappage des sources d'événements Kafka à l'aide de la console ou de l'API Lambda.

Pour configurer le mode provisionné pour un mappage des sources d'événements existant (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez la fonction avec le mappage des sources d'événements pour laquelle vous souhaitez configurer le mode provisionné.
3. Choisissez Configuration, puis Déclencheurs.
4. Choisissez le mappage des sources d'événements pour lequel vous souhaitez configurer le mode alloué, puis choisissez Modifier.
5. Sous Mode provisionné, sélectionnez Configurer.

- Pour le Nombre minimal de sondeurs d'événements, saisissez une valeur comprise entre 1 et 200. Si vous ne spécifiez aucune valeur, Lambda choisit la valeur par défaut 1.
- Pour le Nombre maximal de sondeurs d'événements, saisissez une valeur comprise entre 1 et 2 000. Cette valeur doit être supérieure ou égale à la valeur du Nombre minimal de sondeurs d'événements. Si vous ne spécifiez aucune valeur, Lambda choisit la valeur par défaut 200.

6. Choisissez Enregistrer.

Vous pouvez configurer le mode provisionné par programmation à l'aide de l'ProvisionedPollerConfigobjet de votre. EventSourceMappingConfiguration Par exemple, la commande UpdateEventSourceMappingCLI suivante configure une `MinimumPollers` valeur de 5 et une `MaximumPollers` valeur de 100.

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --provisioned-poller-config '{"MinimumPollers": 5, "MaximumPollers": 100}'
```

Après avoir configuré le mode alloué, vous pouvez observer l'utilisation des sondeurs d'événements pour votre charge de travail en surveillant la métrique `ProvisionedPollers`. Pour de plus amples informations, veuillez consulter the section called "Métriques de mappage des sources d'événements".

Pour désactiver le mode provisionné et revenir au mode par défaut (à la demande), vous pouvez utiliser la commande UpdateEventSourceMappingCLI suivante :

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --provisioned-poller-config '{}'
```

Fonctionnalités avancées de gestion des erreurs et de performance

Pour les mappages de sources d'événements Kafka avec le mode provisionné activé, vous pouvez configurer des fonctionnalités supplémentaires pour améliorer la gestion des erreurs et les performances :

- Configurations de nouvelles tentatives : contrôlez la façon dont Lambda gère les enregistrements ayant échoué avec un maximum de tentatives, des limites d'âge des enregistrements, un fractionnement par lots et des réponses partielles par lots.

- Destinations Kafka en cas de défaillance : envoyez les enregistrements ayant échoué vers une rubrique Kafka pour un traitement ou une analyse ultérieurs.

Bonnes pratiques et considérations lors de l'utilisation du mode provisionné

La configuration optimale du nombre minimal et maximal de sondeurs d'événements pour votre mappage des sources d'événements dépend des exigences de performances de votre application. Nous vous recommandons de commencer avec le nombre minimal de sondeurs d'événements par défaut afin de définir le profil de performances. Ajustez votre configuration en fonction des modèles de traitement des messages observés et du profil de performances souhaité.

Pour les charges de travail associées à des pics de trafic et à des exigences de performances strictes, augmentez le nombre minimal de sondeurs d'événements de manière à gérer les pics soudains de messages. Pour déterminer le nombre minimal de sondeurs d'événements requis, prenez en compte le nombre de messages par seconde de votre charge de travail et la taille moyenne de la charge utile, et utilisez la capacité de débit d'un seul sondeur d'événements (jusqu'à 5 MBps) comme référence.

Pour maintenir un traitement ordonné au sein d'une partition, Lambda limite le nombre maximal de sondeurs d'événements au nombre de partitions dans la rubrique. En outre, le nombre maximal de sondeurs d'événements auxquels votre mappage des sources d'événements peut être mis à l'échelle dépend des paramètres de simultanéité de la fonction.

Lorsque vous activez le mode provisionné, mettez à jour vos paramètres réseau pour supprimer les points de terminaison AWS PrivateLink VPC et les autorisations associées.

Optimisation des coûts pour le mode provisionné

Tarification du mode provisionné

Le mode provisionné est facturé en fonction du nombre minimum de sondeurs d'événements provisionnés et des sondeurs d'événements consommés lors du dimensionnement automatique. Les frais sont calculés à l'aide d'une unité de facturation appelée Event Poller Unit (EPU). Vous payez en fonction du nombre et de la durée d' EPUs utilisation, exprimés en Event-Poller-Unit-hours. Vous pouvez utiliser le mode provisionné avec un seul ESM pour les applications sensibles aux performances ou vous pouvez en regrouper plusieurs au sein d'un même VPC pour partager ESMs la capacité et les coûts de l'EPU. Nous aborderons en détail deux fonctionnalités qui vous aident à optimiser les coûts de votre mode provisionné. Pour plus de détails sur les tarifs, consultez la section Tarification AWS Lambda.

Utilisation améliorée de l'EPU

Chaque EPU prend en charge une capacité de MB/s débit allant jusqu'à 20 pour l'interrogation d'événements et prend en charge 10 sondeurs d'événements par défaut. Lorsque vous créez un mode provisionné pour Kafka ESM en définissant un minimum et un maximum de sondeurs, il utilise un nombre minimum d'interrogateurs pour le provisionnement sur la EPUs base de 10 sondeurs d'événements par EPU par défaut. Cependant, chaque sondeur d'événements peut évoluer indépendamment pour prendre en charge jusqu'à 5 % de la capacité MB/s de débit, ce qui peut nécessiter une plus faible densité de sondeurs d'événements sur un EPU spécifique et peut déclencher une mise à l'échelle de. EPUs Le nombre de sondeurs d'événements alloués sur un EPU dépend de la capacité de calcul consommée par chaque sondeur d'événements. Cette approche d'utilisation améliorée de l'EPU permet aux enquêteurs ayant des exigences de débit variables d'utiliser efficacement la capacité de l'EPU, réduisant ainsi les coûts pour tous. ESMs

Regroupement ESM

Pour optimiser davantage les coûts de votre mode provisionné, vous pouvez regrouper plusieurs Kafka ESMs afin de partager la capacité de l'EPU. Grâce au regroupement ESM et à l'utilisation améliorée de l'EPU, vous pouvez réduire les coûts du mode provisionné jusqu'à 90 % pour les charges de travail à faible débit par rapport à l'exécution en mode ESM unique. Tous ceux ESMs qui nécessitent moins d'une capacité EPU bénéficieront du regroupement ESM. Cela nécessite ESMs généralement un nombre minimal de sondeurs d'événements pour répondre à leurs besoins en matière de débit. Cette fonctionnalité vous permettra d'adopter le mode provisionné pour toutes vos charges de travail Kafka et de bénéficier de fonctionnalités telles que la validation du schéma, le filtrage des Avro/Protobuf événements, les appels à faible latence et la gestion améliorée des erreurs, uniquement disponibles en mode provisionné.

Lorsque vous configurez le `PollerGroupName` paramètre avec la même valeur pour plusieurs éléments d' ESMs un même Amazon VPC, ceux-ci ESMs partagent des ressources EPU au lieu de nécessiter chacun une capacité EPU dédiée. Vous pouvez regrouper jusqu'à 100 ESMs par groupe de sondeurs et le nombre maximum de sondeurs par groupe ne peut pas dépasser 2 000. ESMs

Pour configurer le regroupement ESM (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez votre fonction.
3. Choisissez Configuration, puis Déclencheurs.
4. Lorsque vous créez un nouveau mappage de source d'événements Kafka ou que vous modifiez un mappage existant, sélectionnez Configurer en mode provisionné.

5. Pour le Nombre minimal de sondeurs d'événements, saisissez une valeur comprise entre 1 et 200.
6. Pour le Nombre maximal de sondeurs d'événements, saisissez une valeur comprise entre 1 et 2 000.
7. Pour le nom du groupe Poller, entrez un identifiant pour le groupe. Utilisez le même nom pour les autres ESMs personnes que vous souhaitez regrouper.
8. Choisissez Enregistrer.

Pour configurer le regroupement ESM (AWS CLI)

L'exemple suivant crée un ESM avec un groupe de sondeurs nommé : `production-app-group`

```
aws lambda create-event-source-mapping \
  --function-name myFunction1 \
  --event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/MyCluster/abcd1234 \
  --topics topic1 \
  --starting-position LATEST \
  --provisioned-poller-config '{
    "MinimumPollers": 1,
    "MaximumPollers": 10,
    "PollerGroupName": "production-app-group"
  }'
```

Pour ajouter un autre ESM au même groupe (partageant la capacité de l'EPU), utilisez le même : PollerGroupName

```
aws lambda create-event-source-mapping \
  --function-name myFunction2 \
  --event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/MyCluster/abcd1234 \
  --topics topic2 \
  --starting-position LATEST \
  --provisioned-poller-config '{
    "MinimumPollers": 1,
    "MaximumPollers": 10,
    "PollerGroupName": "production-app-group"
  }'
```

> **Note**
>
> Vous pouvez mettre `PollerGroupName` à jour le pour déplacer un ESM vers un autre groupe, ou supprimer un ESM d'un groupe en transmettant une chaîne vide (« ») pour : `PollerGroupName`

```
# Move ESM to a different group
aws lambda update-event-source-mapping \
  --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
  --provisioned-poller-config '{
    "MinimumPollers": 1,
    "MaximumPollers": 10,
    "PollerGroupName": "new-group-name"
  }'

# Remove ESM from group (use dedicated resources)
aws lambda update-event-source-mapping \
  --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
  --provisioned-poller-config '{
    "MinimumPollers": 1,
    "MaximumPollers": 10,
    "PollerGroupName": ""
  }'
```

Considérations relatives aux stratégies de regroupement

- Limite des applications : groupe ESMs appartenant aux mêmes applications ou services pour une meilleure allocation et une meilleure gestion des coûts. Envisagez d'utiliser des conventions de dénomination telles que `app-name-environment` (par exemple,`order-processor-prod`).
- Schéma de trafic : évitez les regroupements ESMs présentant un débit élevé et des pics de trafic, car cela pourrait entraîner une contention des ressources.
- Rayon d'explosion — Tenez compte de l'impact si l'infrastructure partagée rencontre des problèmes. Tous ESMs les membres d'un même groupe sont concernés par les limites des ressources partagées. Pour les charges de travail critiques, vous pouvez utiliser des groupes distincts ou dédiés. ESMs

Exemple d'optimisation des coûts

Imaginons un scénario dans lequel vous en avez 10 ESMs, chacun étant configuré avec un sondeur d'événements et un débit inférieur à 2 Mo/s :

Sans regroupement :

- Chaque ESM a besoin de son propre EPU
- Total EPUs nécessaire : 10
- Coût par EPU : 0,185 $/heure dans l'est des États-Unis (Virginie du Nord)
- Coût mensuel de l'EPU (720 heures) : 10 × 720 × 0,185$ = 1 332$

Avec regroupement :

- Tous les 10 ESMs partagent la capacité de l'EPU
- 10 sondeurs événementiels peuvent être utilisés dans 1 EPU (avec 10 nouveaux sondeurs par support EPU)
- Total EPUs nécessaire : 1
- Coût mensuel de l'EPU (720 heures) : 1 × 720 × 0,185$ = 133,20$
- Économies de coûts : 90 % (1 198,80$ d'économies par mois)

Positions de départ des interrogations et flux Apache Kafka dans Lambda

Le paramètre StartingPosition indique à Lambda quand commencer à lire les messages depuis votre flux Amazon MSK ou Apache Kafka autogéré. Vous avez le choix entre trois options :

- Dernier : Lambda commence à lire juste après l'enregistrement le plus récent de la rubrique Kafka.
- Supprimer l'horizon : Lambda commence par le dernier enregistrement non découpé de la rubrique Kafka. Il s'agit également de l'enregistrement le plus ancien de la rubrique.
- À l'horodatage : Lambda commence à lire à partir d'une position définie par un horodatage, en secondes Unix. Utilisez le paramètre StartingPositionTimestamp pour spécifier l'horodatage.

L'interrogation des flux lors des mises à jour et de la création du mappage des sources d'événements est finalement cohérente :

- Lors de la création du mappage des sources d'événements, le démarrage de l'interrogation des événements depuis le flux peut prendre plusieurs minutes.
- Lors des mises à jour du mappage des sources d'événements, l'arrêt et le redémarrage de l'interrogation des événements depuis le flux peuvent prendre jusqu'à 90 secondes.

Ce comportement signifie que si vous spécifiez `LATEST` comme position de départ du flux, le mappage des sources d'événements peut manquer des événements lors d'une création ou d'une mise à jour. Pour vous assurer qu'aucun événement n'est manqué, spécifiez `TRIM_HORIZON` ou `AT_TIMESTAMP`.

ID de groupe de consommateurs personnalisable dans Lambda

Lorsque vous configurez Amazon MSK ou Apache Kafka autogéré comme source d'événements, vous pouvez spécifier un ID de groupe de consommateurs. Cet identifiant de groupe de consommateurs est un identifiant existant pour le groupe de clients Kafka auquel vous souhaitez rattacher votre fonction Lambda. Vous pouvez utiliser cette fonction pour migrer facilement toutes les configurations de traitement d'enregistrements Kafka en cours depuis d'autres clients vers Lambda.

Kafka distribue les messages à tous les consommateurs d'un groupe de consommateurs. Si vous spécifiez un ID de groupe de consommateurs qui compte d'autres consommateurs actifs, Lambda ne reçoit qu'une partie des messages de la rubrique Kafka. Si vous souhaitez que Lambda gère tous les messages de la rubrique, désactivez tous les autres consommateurs de ce groupe de consommateurs.

De plus, si vous spécifiez un ID de groupe de consommateurs et que Kafka trouve un groupe de consommateurs existant valide avec le même ID, Lambda ignore le paramètre StartingPosition pour le mappage des sources d'événements. Lambda commence plutôt à traiter les enregistrements en fonction de la compensation engagée par le groupe de consommateurs. Si vous spécifiez un identifiant de groupe de consommateurs et que Kafka ne trouve aucun groupe de consommateurs existant, Lambda configure votre source d'événement avec le spécifié `StartingPosition`.

L'identifiant du groupe de consommateurs que vous spécifiez doit être unique parmi toutes vos sources d'événements Kafka. Après avoir créé un mappage des sources d'événements Kafka avec l'identifiant de groupe de consommateurs spécifié, vous ne pouvez plus mettre à jour cette valeur.

Filtrage des événements provenant d'Amazon MSK et sources d'événements Apache Kafka autogéré

Vous pouvez utiliser le filtrage d'événements pour contrôler les enregistrements d'un flux ou d'une file d'attente que Lambda envoie à votre fonction. Pour obtenir des informations générales sur le fonctionnement du filtrage des événements, consultez the section called "Filtrage des événements".

> **Note**
>
> Les mappages Amazon MSK et Apache Kafka autogéré ne prennent en charge que le filtrage sur la clé `value`.

Rubriques

- Notions de base du filtrage d'événements Lambda

Notions de base du filtrage d'événements Lambda

Supposons qu'un producteur écrive des messages à une rubrique de votre cluster Kafka, au format JSON valide ou sous forme de chaînes simples. Un exemple d'enregistrement ressemblerait à ce qui suit, avec le message converti en chaîne encodée en Base64 dans le champ `value`.

```
{
    "mytopic-0":[
        {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[]
        }
    ]
}
```

Supposons que votre producteur Apache Kafka écrive des messages dans votre rubrique au format JSON suivant.

```
{
    "device_ID": "AB1234",
    "session":{
        "start_time": "yyyy-mm-ddThh:mm:ss",
        "duration": 162
    }
}
```

Vous pouvez utiliser la clé `value` pour filtrer les enregistrements. Supposons que vous vouliez filtrer uniquement les enregistrements où `device_ID` commence par les lettres AB. L'objet `FilterCriteria` serait le suivant.

```
{
    "Filters": [
        {
            "Pattern": "{ \"value\" : { \"device_ID\" : [ { \"prefix\": \"AB\" } ] } }"
        }
    ]
}
```

Pour plus de clarté, voici la valeur du `Pattern` de filtre étendu en JSON simple :

```
{
    "value": {
        "device_ID": [ { "prefix": "AB" } ]
      }
}
```

Vous pouvez ajouter votre filtre à l'aide de la console, d'AWS CLI ou d'un modèle AWS SAM.

Console

Pour ajouter ce filtre à l'aide de la console, suivez les instructions de Attacher des critères de filtre à un mappage de sources d'événements (console) et saisissez la chaîne suivante comme critère de filtre.

```
{ "value" : { "device_ID" : [ { "prefix":  "AB" } ] } }
```

AWS CLI

Pour créer un nouveau mappage des sources d'événements avec ces critères de filtrage à l'aide de l'outil AWS Command Line Interface (AWS CLI), exécutez la commande suivante.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:kafka:us-east-2:123456789012:cluster/my-cluster/
b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : { \"device_ID\" :
 [ { \"prefix\":  \"AB\" } ] } }"}]}'
```

Pour ajouter ces critères de filtre à un mappage des sources d'événements existant, exécutez la commande suivante.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : { \"device_ID\" :
 [ { \"prefix\":  \"AB\" } ] } }"}]}'
```

AWS SAM

Pour ajouter ce filtre à l'aide d'AWS SAM, ajoutez l'extrait suivant au modèle YAML de votre source d'événement.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "value" : { "device_ID" : [ { "prefix":  "AB" } ] } }'
```

Avec Kafka, vous pouvez également filtrer les enregistrements dont le message est une chaîne simple. Supposons que vous vouliez ignorer les messages dont la chaîne est « error ». L'objet `FilterCriteria` se présente comme suit.

```
{
    "Filters": [
        {
            "Pattern": "{ \"value\" : [ { \"anything-but\": [ \"error\" ] } ] }"
        }
    ]
```

```
}
```

Pour plus de clarté, voici la valeur du `Pattern` de filtre étendu en JSON simple :

```
{
    "value": [
        {
        "anything-but": [ "error" ]
        }
    ]
}
```

Vous pouvez ajouter votre filtre à l'aide de la console, d'AWS CLI ou d'un modèle AWS SAM.

Console

Pour ajouter ce filtre à l'aide de la console, suivez les instructions de Attacher des critères de filtre à un mappage de sources d'événements (console) et saisissez la chaîne suivante comme critère de filtre.

```
{ "value" : [ { "anything-but": [ "error" ] } ] }
```

AWS CLI

Pour créer un nouveau mappage des sources d'événements avec ces critères de filtrage à l'aide de l'outil AWS Command Line Interface (AWS CLI), exécutez la commande suivante.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:kafka:us-east-2:123456789012:cluster/my-cluster/
b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : [ { \"anything-but\":
 [ \"error\" ] } ] }"}]}'
```

Pour ajouter ces critères de filtre à un mappage des sources d'événements existant, exécutez la commande suivante.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : [ { \"anything-but\":
 [ \"error\" ] } ] }"}]}'
```

AWS SAM

Pour ajouter ce filtre à l'aide d'AWS SAM, ajoutez l'extrait suivant au modèle YAML de votre source d'événement.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "value" : [ { "anything-but": [ "error" ] } ] }'
```

Les messages Kafka doivent être des chaînes codées en UTF-8, soit des chaînes en texte brut, soit au format JSON. En effet, Lambda décode les tableaux d'octets Kafka en UTF-8 avant d'appliquer des critères de filtre. Si vos messages utilisent un autre encodage, tel que UTF-16 ou ASCII, ou si le format du message ne correspond pas au format `FilterCriteria`, Lambda traite uniquement les filtres de métadonnées. Le tableau suivant résume le comportement spécifique :

Format du du message entrant	Modèle de filtre de format pour les propriétés des messages	Action obtenue.
Chaîne de texte brut	Chaîne de texte brut	Lambda filtre en fonction de vos critères de filtre.
Chaîne de texte brut	Pas de modèle de filtre pour les propriétés des données	Lambda filtre (uniquement selon les autres propriétés de métadonnées) en fonction de vos critères de filtre.
Chaîne de texte brut	JSON valide	Lambda filtre (uniquement selon les autres propriétés de métadonnées) en fonction de vos critères de filtre.
JSON valide	Chaîne de texte brut	Lambda filtre (uniquement selon les autres propriétés de métadonnées) en fonction de vos critères de filtre.
JSON valide	Pas de modèle de filtre pour les propriétés des données	Lambda filtre (uniquement selon les autres propriétés de

Format du du message entrant	Modèle de filtre de format pour les propriétés des messages	Action obtenue.
		métadonnées) en fonction de vos critères de filtre.
JSON valide	JSON valide	Lambda filtre en fonction de vos critères de filtre.
Chaîne non codée UTF-8	JSON, chaîne de texte brut ou aucun modèle	Lambda filtre (uniquement selon les autres propriétés de métadonnées) en fonction de vos critères de filtre.

Utilisation de registres de schémas avec les sources d'événements Kafka dans Lambda

Les registres de schémas vous aident à définir et gérer des schémas de flux de données. Un schéma définit la structure et le format d'un enregistrement de données. Dans le contexte des mappages des sources d'événements Kafka, vous pouvez configurer un registre de schémas pour valider la structure et le format des messages Kafka par rapport à des schémas prédéfinis avant qu'ils atteignent votre fonction Lambda. Cela ajoute une couche de gouvernance des données à votre application et vous permet de gérer efficacement les formats de données, de garantir la conformité des schémas et d'optimiser les coûts grâce au filtrage des événements.

Cette fonctionnalité est compatible avec tous les langages de programmation, mais tenez compte des points importants suivants :

- Powertools for Lambda fournit un support spécifique pour Java, Python TypeScript et assure la cohérence avec les modèles de développement Kafka existants et permet un accès direct aux objets métier sans code de désérialisation personnalisé
- Cette fonctionnalité n'est disponible que pour les mappages des sources d'événements utilisant le mode provisionné. Le registre des schémas ne prend pas en charge les mappages des sources d'événements en mode à la demande. Si vous utilisez le mode provisionné et qu'un registre de schéma est configuré, vous ne pouvez pas passer en mode à la demande, sauf si vous supprimez d'abord la configuration de votre registre de schémas. Pour de plus amples informations, consultez the section called "Mode alloué".
- Vous ne pouvez configurer qu'un seul registre de schémas par mappage des sources d'événements (ESM). L'utilisation d'un registre de schémas avec votre source d'événements Kafka peut augmenter votre utilisation d'unités de sondeur d'événements (EPU) Lambda, une dimension tarifaire pour le mode provisionné.

Rubriques

- Options du registre de schémas
- Comment Lambda effectue la validation du schéma pour les messages Kafka
- Configuration d'un registre de schémas Kafka
- Filtrage pour Avro et Protobuf
- Formats de données utiles et comportement de désérialisation
- Utilisation de données désérialisées dans des fonctions Lambda

- Méthodes d'authentification pour votre registre de schémas
- Gestion des erreurs et résolution des problèmes liés au registre de schémas

Options du registre de schémas

Lambda prend en charge les options de registre de schémas suivantes :

- AWS Glue Registre des schémas
- Registre de schémas Confluent Cloud
- Registre de schémas Confluent autogéré

Votre registre de schémas prend en charge la validation des messages dans les formats de données suivants :

- Apache Avro
- Protocol Buffers (Protobuf)
- Schéma JSON (JSON-SE)

Pour utiliser un registre de schémas, assurez-vous d'abord que votre mappage des sources d'événements est en mode provisionné. Lorsque vous utilisez un registre de schémas, Lambda ajoute des métadonnées relatives au schéma aux données utiles. Pour plus d'informations, consultez Formats de données utiles et comportement de désérialisation.

Comment Lambda effectue la validation du schéma pour les messages Kafka

Lorsque vous configurez un registre de schémas, Lambda exécute les étapes suivantes pour chaque message Kafka :

1. Lambda interroge l'enregistrement Kafka de votre cluster.
2. Lambda valide les attributs de message sélectionnés dans l'enregistrement par rapport à un schéma spécifique de votre registre de schémas.
   - Si le schéma associé au message n'est pas trouvé dans le registre, Lambda envoie le message à une DLQ avec le code de motif `SCHEMA_NOT_FOUND`.
3. Lambda désérialise le message conformément à la configuration du registre de schémas pour valider le message. Si le filtrage des événements est configuré, Lambda effectue ensuite le filtrage en fonction des critères de filtre configurés.

- Si la désérialisation échoue, Lambda envoie le message à une DLQ avec le code de motif `DESERIALIZATION_ERROR`. Si aucune DLQ n'est configurée, Lambda abandonne le message.

4. Si le message est validé par le registre de schémas et qu'il n'est pas filtré selon vos critères de filtre, Lambda invoque votre fonction avec le message.

Cette fonctionnalité est destinée à valider les messages déjà produits à l'aide de clients Kafka intégrés à un registre de schémas. Nous vous recommandons de configurer vos producteurs Kafka pour qu'ils fonctionnent avec votre registre de schémas afin de créer des messages correctement formatés.

Configuration d'un registre de schémas Kafka

Les étapes de console suivantes ajoutent une configuration de registre de schémas Kafka à votre mappage des sources d'événements.

Ajouter une configuration de registre de schémas Kafka à votre mappage des sources d'événements (console)

1. Ouvrez la page Fonction de la console Lambda.
2. Choisissez Configuration.
3. Choisissez Déclencheurs.
4. Sélectionnez le mappage des sources d'événements Kafka pour lequel vous souhaitez configurer un registre de schémas, puis choisissez Modifier.
5. Sous Configuration du sondeur d'événements, choisissez Configurer le registre de schémas. Votre mappage des sources d'événements doit être en mode provisionné pour voir cette option.
6. Pour l'URI du registre de schéma, entrez l'ARN de votre registre de AWS Glue schémas ou l'URL HTTPS de votre registre de schémas Confluent Cloud ou de votre registre de schémas Confluent autogéré.
7. Les étapes de configuration suivantes indiquent à Lambda comment accéder à votre registre de schémas. Pour de plus amples informations, veuillez consulter ???.

   - Pour Type de configuration d'accès, choisissez le type d'authentification utilisé par Lambda pour accéder à votre registre de schémas.
   - Pour URI de configuration d'accès, entrez l'ARN du secret Secrets Manager pour vous authentifier auprès de votre registre de schémas, le cas échéant. Assurez-vous que le rôle d'exécution de votre fonction contient les autorisations appropriées.

8. Le champ Chiffrement s'applique uniquement si votre registre de schémas est signé par une autorité de certification (CA) privée ou une autorité de certification (CA) qui ne figure pas dans le magasin d'approbations Lambda. Le cas échéant, fournissez la clé secrète contenant le certificat CA privé utilisé par votre registre de schémas pour le chiffrement TLS.
9. Pour Format d'enregistrement de l'événement, choisissez la manière dont vous souhaitez que Lambda transmette les enregistrements à votre fonction après la validation du schéma. Pour plus d'informations consultez Exemples de formats de données utiles.

    - Si vous choisissez JSON, Lambda fournit les attributs que vous sélectionnez dans Attribut de validation du schéma ci-dessous au format JSON standard. Pour les attributs que vous ne sélectionnez pas, Lambda les fournit en l'état.
    - Si vous choisissez SOURCE, Lambda fournit les attributs que vous avez sélectionnés dans Attribut de validation du schéma ci-dessous dans leur format source d'origine.
10. Pour Attribut de validation du schéma, sélectionnez les attributs de message que Lambda doit valider et désérialiser à l'aide de votre registre de schémas. Vous devez sélectionner au moins une option entre KEY et VALUE. Si vous avez choisi JSON pour le format d'enregistrement d'événement, Lambda désérialise également les attributs de message sélectionnés avant de les envoyer à votre fonction. Pour plus d'informations, consultez Formats de données utiles et comportement de désérialisation.
11. Choisissez Enregistrer.

Vous pouvez également utiliser l'API Lambda pour créer ou mettre à jour votre mappage des sources d'événements avec une configuration de registre de schémas. Les exemples suivants montrent comment configurer un registre de schéma AWS Glue ou un registre de schéma Confluent à l'aide du AWS CLI, qui correspond aux opérations d'CreateEventSourceMappingAPI UpdateEventSourceMappinget de la référence AWS Lambda d'API :

> **Important**
>
> Si vous mettez à jour un champ de configuration du registre de schéma à l'aide de l'`update-event-source-mapping`API AWS CLI ou de l'API, vous devez mettre à jour tous les champs de configuration du registre de schéma.

Create Event Source Mapping

```
aws lambda create-event-source-mapping \
```

```
  --function-name my-schema-validator-function \
  --event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/my-cluster/
a1b2c3d4-5678-90ab-cdef-11111EXAMPLE \
  --topics my-kafka-topic \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=1 \
  --amazon-managed-kafka-event-source-mapping '{
      "SchemaRegistryConfig" : {
          "SchemaRegistryURI": "https://abcd-ef123.us-west-2.aws.confluent.cloud",
          "AccessConfigs": [{
              "Type": "BASIC_AUTH",
              "URI": "arn:aws:secretsmanager:us-
east-1:123456789012:secret:secretName"
          }],
          "EventRecordFormat": "JSON",
          "SchemaValidationConfigs": [
          {
              "Attribute": "KEY"
          },
          {
              "Attribute": "VALUE"
          }]
      }
  }'
```

Update AWS Glue Schema Registry

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --amazon-managed-kafka-event-source-mapping '{
        "SchemaRegistryConfig" : {
            "SchemaRegistryURI": "arn:aws:glue:us-east-1:123456789012:registry/
registryName",
            "EventRecordFormat": "JSON",
            "SchemaValidationConfigs": [
            {
                "Attribute": "KEY"
            },
            {
                "Attribute": "VALUE"
            }]
        }
    }'
```

Update Confluent Schema Registry with Authentication

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --amazon-managed-kafka-event-source-mapping '{
        "SchemaRegistryConfig" : {
            "SchemaRegistryURI": "https://abcd-ef123.us-west-2.aws.confluent.cloud",
            "AccessConfigs": [{
                "Type": "BASIC_AUTH",
                "URI": "arn:aws:secretsmanager:us-
east-1:123456789012:secret:secretName"
            }],
            "EventRecordFormat": "JSON",
            "SchemaValidationConfigs": [
            {
                "Attribute": "KEY"
            },
            {
                "Attribute": "VALUE"
            }]
        }
    }'
```

Update Confluent Schema Registry without Authentication

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --amazon-managed-kafka-event-source-mapping '{
        "SchemaRegistryConfig" : {
            "SchemaRegistryURI": "https://abcd-ef123.us-west-2.aws.confluent.cloud",
            "EventRecordFormat": "JSON",
            "SchemaValidationConfigs": [
            {
                "Attribute": "KEY"
            },
            {
                "Attribute": "VALUE"
            }]
        }
    }'
```

Remove Schema Registry Configuration

Pour supprimer une configuration de registre de schéma de votre mappage de source d'événements, vous pouvez utiliser la commande CLI UpdateEventSourceMappingdans la référence AWS Lambda d'API.

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --amazon-managed-kafka-event-source-mapping '{
        "SchemaRegistryConfig" : {}
    }'
```

Filtrage pour Avro et Protobuf

Lorsque vous utilisez les formats Avro ou Protobuf avec un registre de schémas, vous pouvez appliquer le filtrage des événements à votre fonction Lambda. Les modèles de filtre sont appliqués à la représentation JSON classique désérialisée de vos données après validation du schéma. Par exemple, avec un schéma Avro définissant les détails du produit, y compris le prix, vous pouvez filtrer les messages en fonction de la valeur du prix :

> **Note**
>
> Lors de la désérialisation, l'objet Avro est converti en JSON standard, ce qui signifie qu'il ne peut pas être reconverti directement en objet Avro. Si vous devez effectuer une conversion en objet Avro, utilisez plutôt le format SOURCE.
> Pour la désérialisation de Protobuf, les noms de champs dans le JSON obtenu correspondent à ceux définis dans votre schéma, plutôt que d'être convertis en casse mixte comme le fait généralement Protobuf. Gardez cela à l'esprit lorsque vous créez des schémas de filtrage.

```
aws lambda create-event-source-mapping \
    --function-name myAvroFunction \
    --topics myAvroTopic \
    --starting-position TRIM_HORIZON \
    --kafka-bootstrap-servers '["broker1:9092", "broker2:9092"]' \
    --schema-registry-config '{
        "SchemaRegistryURI": "arn:aws:glue:us-east-1:123456789012:registry/
myAvroRegistry",
        "EventRecordFormat": "JSON",
```

```
        "SchemaValidationConfigs": [
            {
                "Attribute": "VALUE"
            }
        ]
    }' \
    --filter-criteria '{
        "Filters": [
            {
                "Pattern": "{ \"value\" : { \"field_1\" : [\"value1\"], \"field_2\" :
[\"value2\"] } }"
            }
        ]
    }'
```

Dans cet exemple, le modèle de filtre analyse l'objet `value` en faisant correspondre les messages dans `field_1` avec `"value1"`, et `field_2` avec `"value2"`. Les critères de filtre sont évalués par rapport aux données désérialisées, une fois que Lambda a converti le message du format Avro en JSON.

Pour plus d'informations sur le filtrage des événements, consultez la section Filtrage des événements Lambda.

Formats de données utiles et comportement de désérialisation

Lorsque vous utilisez un registre de schémas, Lambda fournit les données utiles finales à votre fonction dans un format similaire aux données utiles d'événement ordinaires, avec quelques champs supplémentaires. Les champs supplémentaires dépendent du paramètre `SchemaValidationConfigs`. Pour chaque attribut que vous sélectionnez pour validation (clé ou valeur), Lambda ajoute les métadonnées de schéma correspondantes aux données utiles.

> **Note**
>
> Vous devez passer aws-lambda-java-eventsà la version 3.16.0 ou supérieure pour utiliser les champs de métadonnées du schéma.

Par exemple, si vous validez le champ `value`, Lambda ajoute un champ appelé `valueSchemaMetadata` à vos données utiles. De même, pour le champ `key`, Lambda ajoute un champ appelé `keySchemaMetadata`. Ces métadonnées contiennent des informations sur le format des données et l'ID de schéma utilisés pour la validation :

```
"valueSchemaMetadata": {
    "dataFormat": "AVRO",
    "schemaId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
}
```

Le paramètre `EventRecordFormat` peut être défini sur `JSON` ou `SOURCE`, ce qui détermine la manière dont Lambda gère les données validées par le schéma avant de les transmettre à votre fonction. Chaque option offre des capacités de traitement différentes :

- `JSON` : Lambda désérialise les attributs validés au format JSON standard, ce qui rend les données prêtes à être utilisées directement dans les langages prenant en charge le format JSON natif. Ce format est idéal lorsque vous n'avez pas besoin de conserver le format binaire d'origine ou de travailler avec les classes générées.
- `SOURCE` : Lambda préserve le format binaire original des données sous forme de chaîne codée en Base64, ce qui permet une conversion directe en objets Avro ou Protobuf. Ce format est essentiel lorsque vous travaillez avec des langages fortement typés ou lorsque vous devez conserver toutes les fonctionnalités des schémas Avro ou Protobuf.

Sur la base de ces caractéristiques de format et des considérations spécifiques au langage, nous recommandons les formats suivants :

Formats recommandés selon le langage de programmation

Language	Avro	Protobuf	JSON
Java	SOURCE	SOURCE	SOURCE
Python	JSON	JSON	JSON
NodeJS	JSON	JSON	JSON
.NET	SOURCE	SOURCE	SOURCE
Autres	JSON	JSON	JSON

Les sections suivantes décrivent ces formats en détail et fournissent des exemples de données utiles pour chaque format.

Format JSON

Si vous choisissez `JSON` comme tel`EventRecordFormat`, Lambda valide et désérialise les attributs de message que vous avez sélectionnés dans le `SchemaValidationConfigs` champ (les attributs). `key` and/or `value` Lambda fournit ces attributs sélectionnés sous forme de chaînes codées en Base64 de leur représentation JSON standard dans votre fonction.

> **Note**
>
> Lors de la désérialisation, l'objet Avro est converti en JSON standard, ce qui signifie qu'il ne peut pas être reconverti directement en objet Avro. Si vous devez effectuer une conversion en objet Avro, utilisez plutôt le format SOURCE.
> Pour la désérialisation de Protobuf, les noms de champs dans le JSON obtenu correspondent à ceux définis dans votre schéma, plutôt que d'être convertis en casse mixte comme le fait généralement Protobuf. Gardez cela à l'esprit lorsque vous créez des schémas de filtrage.

Voici un exemple de données utiles, en supposant que vous choisissez `JSON` comme `EventRecordFormat` et les attributs `key` et `value` comme `SchemaValidationConfigs` :

```
{
   "eventSource":"aws:kafka",
   "eventSourceArn":"arn:aws:kafka:us-east-1:123456789012:cluster/vpc-2priv-2pub/
a1b2c3d4-5678-90ab-cdef-EXAMPLE11111-1",
   "bootstrapServers":"b-2.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092",
   "records":{
      "mytopic-0":[
         {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "key":"abcDEFghiJKLmnoPQRstuVWXyz1234==", //Base64 encoded string of JSON
            "keySchemaMetadata": {
                "dataFormat": "AVRO",
                "schemaId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
            },
            "value":"abcDEFghiJKLmnoPQRstuVWXyz1234", //Base64 encoded string of JSON
```

```
            "valueSchemaMetadata": {
                "dataFormat": "AVRO",
                "schemaId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
            },
            "headers":[
               {
                  "headerKey":[
                     104,
                     101,
                     97,
                     100,
                     101,
                     114,
                     86,
                     97,
                     108,
                     117,
                     101
                  ]
               }
            ]
         }
      ]
   }
}
```

Dans cet exemple :

- `key` et `value` sont des chaînes codées en Base64 de leur représentation JSON après désérialisation.
- Lambda inclut des métadonnées de schéma pour les deux attributs dans `keySchemaMetadata` et `valueSchemaMetadata`.
- Votre fonction peut décoder les chaînes `key` et `value` pour accéder aux données JSON désérialisées.

Le format JSON est recommandé pour les langages qui ne sont pas fortement typés, comme Python ou Node.js. Ces langages offrent la prise en charge native de la conversion de JSON en objets.

Format source

Si vous choisissez SOURCE comme EventRecordFormat, Lambda valide toujours l'enregistrement par rapport au registre de schémas, mais fournit les données binaires d'origine à votre fonction sans désérialisation. Ces données binaires sont fournies sous la forme d'une chaîne codée en Base64 des données d'octets d'origine, les métadonnées ajoutées par le producteur étant supprimées. Par conséquent, vous pouvez directement convertir les données binaires brutes en objets Avro et Protobuf dans le code de votre fonction. Nous vous recommandons d'utiliser Powertools for AWS Lambda, qui désérialisera les données binaires brutes et vous fournira directement les objets Avro et Protobuf.

Par exemple, si vous configurez Lambda pour valider à la fois les attributs key et value, mais que vous utilisez le format SOURCE, votre fonction reçoit des données utiles semblables à celles-ci :

```
{
    "eventSource": "aws:kafka",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/vpc-2priv-2pub/
a1b2c3d4-5678-90ab-cdef-EXAMPLE11111-1",
    "bootstrapServers": "b-2.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092",
    "records": {
        "mytopic-0": [
            {
                "topic": "mytopic",
                "partition": 0,
                "offset": 15,
                "timestamp": 1545084650987,
                "timestampType": "CREATE_TIME",
                "key": "abcDEFghiJKLmnoPQRstuVWXyz1234==", // Base64 encoded string of
 Original byte data, producer-appended metadata removed
                "keySchemaMetadata": {
                    "dataFormat": "AVRO",
                    "schemaId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
                },
                "value": "abcDEFghiJKLmnoPQRstuVWXyz1234==", // Base64 encoded string
 of Original byte data, producer-appended metadata removed
                "valueSchemaMetadata": {
                    "dataFormat": "AVRO",
                    "schemaId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
                },
                "headers": [
```

```
                    {
                        "headerKey": [
                            104,
                            101,
                            97,
                            100,
                            101,
                            114,
                            86,
                            97,
                            108,
                            117,
                            101
                        ]
                    }
                ]
            }
        ]
    }
}
```

Dans cet exemple :

- `key` et `value` contiennent tous deux les données binaires d'origine sous forme de chaînes codées en Base64.
- Votre fonction doit gérer la désérialisation à l'aide des bibliothèques appropriées.

Il est recommandé de choisir `SOURCE` pour `EventRecordFormat` si vous utilisez des objets générés par Avro ou Protobuf, en particulier avec des fonctions Java. Cela est dû au fait que Java est fortement typé et nécessite des désérialiseurs spécifiques pour les formats Avro et Protobuf. Dans le code de votre fonction, vous pouvez utiliser votre bibliothèque Avro ou Protobuf préférée pour désérialiser les données.

Utilisation de données désérialisées dans des fonctions Lambda

Powertools for vous AWS Lambda aide à désérialiser les enregistrements Kafka dans votre code de fonction en fonction du format que vous utilisez. Cet utilitaire simplifie l'utilisation des enregistrements Kafka en gérant la conversion des données et en fournissant des ready-to-use objets.

Pour utiliser Powertools AWS Lambda dans votre fonction, vous devez ajouter Powertools en tant que couche AWS Lambda ou en tant que dépendance lors de la création de votre fonction Lambda.

Pour obtenir des instructions de configuration et plus d'informations, consultez les Powertools pour obtenir AWS Lambda la documentation correspondant à votre langue préférée :

- Powertools pour AWS Lambda (Java)
- Powertools pour AWS Lambda (Python)
- Outils électriques pour AWS Lambda () TypeScript
- Powertools pour AWS Lambda (.NET)

> **Note**
>
> Lorsque vous travaillez avec l'intégration du registre de schémas, vous pouvez choisir le format `SOURCE` ou `JSON`. Chaque option prend en charge différents formats de sérialisation, comme indiqué ci-dessous :
>
> | Format | Prend en charge |
> | --- | --- |
> | SOURCE | Avro et Protobuf (en utilisant l'intégration du registre de schémas Lambda) |
> | JSON | Données JSON |

Lorsque vous utilisez le `JSON` format `SOURCE` or, vous pouvez utiliser Powertools pour vous aider AWS à désérialiser les données de votre code de fonction. Voici des exemples de gestion de différents formats de données :

AVRO

Exemple Java :

```
package org.demo.kafka;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.demo.kafka.avro.AvroProduct;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import software.amazon.lambda.powertools.kafka.Deserialization;
import software.amazon.lambda.powertools.kafka.DeserializationType;
import software.amazon.lambda.powertools.logging.Logging;

public class AvroDeserializationFunction implements
 RequestHandler<ConsumerRecords<String, AvroProduct>, String> {

    private static final Logger LOGGER =
 LoggerFactory.getLogger(AvroDeserializationFunction.class);

    @Override
    @Logging
    @Deserialization(type = DeserializationType.KAFKA_AVRO)
    public String handleRequest(ConsumerRecords<String, AvroProduct> records,
 Context context) {
        for (ConsumerRecord<String, AvroProduct> consumerRecord : records) {
            LOGGER.info("ConsumerRecord: {}", consumerRecord);

            AvroProduct product = consumerRecord.value();
            LOGGER.info("AvroProduct: {}", product);

            String key = consumerRecord.key();
            LOGGER.info("Key: {}", key);
        }

        return "OK";
    }

}
```

Exemple Python :

```
from aws_lambda_powertools.utilities.kafka_consumer.kafka_consumer import
 kafka_consumer
from aws_lambda_powertools.utilities.kafka_consumer.schema_config import
 SchemaConfig
from aws_lambda_powertools.utilities.kafka_consumer.consumer_records import
 ConsumerRecords

from aws_lambda_powertools.utilities.typing import LambdaContext
```

```
from aws_lambda_powertools import Logger

logger = Logger(service="kafkaConsumerPowertools")

value_schema_str = open("customer_profile.avsc", "r").read()

schema_config = SchemaConfig(
value_schema_type="AVRO",
value_schema=value_schema_str)

@kafka_consumer(schema_config=schema_config)
def lambda_handler(event: ConsumerRecords, context:LambdaContext):

  for record in event.records:
      value = record.value
      logger.info(f"Received value: {value}")
```

TypeScript exemple :

```
import { kafkaConsumer } from '@aws-lambda-powertools/kafka';

import type { ConsumerRecords } from '@aws-lambda-powertools/kafka/types';
import { Logger } from '@aws-lambda-powertools/logger';
import type { Context } from 'aws-lambda';

const logger = new Logger();

type Value = {
    id: number;
    name: string;
    price: number;
};

const schema = '{
    "type": "record",
    "name": "Product",
    "fields": [
        { "name": "id", "type": "int" },
        { "name": "name", "type": "string" },
        { "name": "price", "type": "double" }
    ]
}';
```

```
export const handler = kafkaConsumer<string, Value>(
    (event: ConsumerRecords<string, Value>, _context: Context) => {
        for (const record of event.records) {
            logger.info(Processing record with key: ${record.key});
            logger.info(Record value: ${JSON.stringify(record.value)});
            // You can add more processing logic here
        }
    },
    {
        value: {
            type: 'avro',
            schema: schema,
        },
    }
);
```

Exemple .NET :

```
using Amazon.Lambda.Core;
using AWS.Lambda.Powertools.Kafka;
using AWS.Lambda.Powertools.Kafka.Avro;
using AWS.Lambda.Powertools.Logging;
using Com.Example;

// Assembly attribute to enable the Lambda function's Kafka event to be converted
 into a .NET class.
[assembly: LambdaSerializer(typeof(PowertoolsKafkaAvroSerializer))]

namespace ProtoBufClassLibrary;

public class Function
{
    public string FunctionHandler(ConsumerRecords<string, CustomerProfile> records,
 ILambdaContext context)
    {
        foreach (var record in records)
        {
            Logger.LogInformation("Processing messagem from topic: {topic}",
 record.Topic);
            Logger.LogInformation("Partition: {partition}, Offset: {offset}",
 record.Partition, record.Offset);
            Logger.LogInformation("Produced at: {timestamp}", record.Timestamp);
```

```
            foreach (var header in record.Headers.DecodedValues())
            {
                Logger.LogInformation($"{header.Key}: {header.Value}");
            }

            Logger.LogInformation("Processing order for: {fullName}",
 record.Value.FullName);
        }

        return "Processed " + records.Count() + " records";
    }
}
```

PROTOBUF

Exemple Java :

```
package org.demo.kafka;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.demo.kafka.protobuf.ProtobufProduct;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import software.amazon.lambda.powertools.kafka.Deserialization;
import software.amazon.lambda.powertools.kafka.DeserializationType;
import software.amazon.lambda.powertools.logging.Logging;

public class ProtobufDeserializationFunction
        implements RequestHandler<ConsumerRecords<String, ProtobufProduct>, String>
 {

    private static final Logger LOGGER =
 LoggerFactory.getLogger(ProtobufDeserializationFunction.class);

    @Override
    @Logging
    @Deserialization(type = DeserializationType.KAFKA_PROTOBUF)
    public String handleRequest(ConsumerRecords<String, ProtobufProduct> records,
 Context context) {
```

```
        for (ConsumerRecord<String, ProtobufProduct> consumerRecord : records) {
            LOGGER.info("ConsumerRecord: {}", consumerRecord);

            ProtobufProduct product = consumerRecord.value();
            LOGGER.info("ProtobufProduct: {}", product);

            String key = consumerRecord.key();
            LOGGER.info("Key: {}", key);
        }

        return "OK";
    }

}
```

Exemple Python :

```
from aws_lambda_powertools.utilities.kafka_consumer.kafka_consumer import
 kafka_consumer

from aws_lambda_powertools.utilities.kafka_consumer.schema_config import
 SchemaConfig
from aws_lambda_powertools.utilities.kafka_consumer.consumer_records import
 ConsumerRecords

from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools import Logger

from user_pb2 import User # protobuf generated class

logger = Logger(service="kafkaConsumerPowertools")

schema_config = SchemaConfig(
value_schema_type="PROTOBUF",
value_schema=User)

@kafka_consumer(schema_config=schema_config)
def lambda_handler(event: ConsumerRecords, context:LambdaContext):

  for record in event.records:
      value = record.value
      logger.info(f"Received value: {value}")
```

TypeScript exemple :

```
import { kafkaConsumer } from '@aws-lambda-powertools/kafka';
import type { ConsumerRecords } from '@aws-lambda-powertools/kafka/types';
import { Logger } from '@aws-lambda-powertools/logger';
import type { Context } from 'aws-lambda';
import { Product } from './product.generated.js';

const logger = new Logger();

type Value = {
    id: number;
    name: string;
    price: number;
};

export const handler = kafkaConsumer<string, Value>(
    (event: ConsumerRecords<string, Value>, _context: Context) => {
        for (const record of event.records) {
            logger.info(Processing record with key: ${record.key});
            logger.info(Record value: ${JSON.stringify(record.value)});
        }
    },
    {
        value: {
            type: 'protobuf',
            schema: Product,
        },
    }
);
```

Exemple .NET :

```
using Amazon.Lambda.Core;
using AWS.Lambda.Powertools.Kafka;
using AWS.Lambda.Powertools.Kafka.Protobuf;
using AWS.Lambda.Powertools.Logging;
using Com.Example;

// Assembly attribute to enable the Lambda function's Kafka event to be converted
 into a .NET class.
[assembly: LambdaSerializer(typeof(PowertoolsKafkaProtobufSerializer))]
```

```
namespace ProtoBufClassLibrary;

public class Function
{
    public string FunctionHandler(ConsumerRecords<string, CustomerProfile> records,
 ILambdaContext context)
    {
        foreach (var record in records)
        {
            Logger.LogInformation("Processing messagem from topic: {topic}",
 record.Topic);
            Logger.LogInformation("Partition: {partition}, Offset: {offset}",
 record.Partition, record.Offset);
            Logger.LogInformation("Produced at: {timestamp}", record.Timestamp);

            foreach (var header in record.Headers.DecodedValues())
            {
                Logger.LogInformation($"{header.Key}: {header.Value}");
            }

            Logger.LogInformation("Processing order for: {fullName}",
 record.Value.FullName);
        }

        return "Processed " + records.Count() + " records";
    }
}
```

JSON

Exemple Java :

```
package org.demo.kafka;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import software.amazon.lambda.powertools.kafka.Deserialization;
import software.amazon.lambda.powertools.kafka.DeserializationType;
```

```
import software.amazon.lambda.powertools.logging.Logging;

public class JsonDeserializationFunction implements
 RequestHandler<ConsumerRecords<String, Product>, String> {

    private static final Logger LOGGER =
 LoggerFactory.getLogger(JsonDeserializationFunction.class);

    @Override
    @Logging
    @Deserialization(type = DeserializationType.KAFKA_JSON)
    public String handleRequest(ConsumerRecords<String, Product> consumerRecords,
 Context context) {
        for (ConsumerRecord<String, Product> consumerRecord : consumerRecords) {
            LOGGER.info("ConsumerRecord: {}", consumerRecord);

            Product product = consumerRecord.value();
            LOGGER.info("Product: {}", product);

            String key = consumerRecord.key();
            LOGGER.info("Key: {}", key);
        }

        return "OK";
    }
}
```

Exemple Python :

```
from aws_lambda_powertools.utilities.kafka_consumer.kafka_consumer import
 kafka_consumer

from aws_lambda_powertools.utilities.kafka_consumer.schema_config import
 SchemaConfig
from aws_lambda_powertools.utilities.kafka_consumer.consumer_records import
 ConsumerRecords

from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools import Logger

logger = Logger(service="kafkaConsumerPowertools")

schema_config = SchemaConfig(value_schema_type="JSON")
```

```
@kafka_consumer(schema_config=schema_config)
def lambda_handler(event: ConsumerRecords, context:LambdaContext):

  for record in event.records:
      value = record.value
      logger.info(f"Received value: {value}")
```

TypeScript exemple :

```
import { kafkaConsumer } from '@aws-lambda-powertools/kafka';
import type { ConsumerRecords } from '@aws-lambda-powertools/kafka/types';
import { Logger } from '@aws-lambda-powertools/logger';
import type { Context } from 'aws-lambda';

const logger = new Logger();

type Value = {
    id: number;
    name: string;
    price: number;
};

export const handler = kafkaConsumer<string, Value>(
    (event: ConsumerRecords<string, Value>, _context: Context) => {
        for (const record of event.records) {
            logger.info(Processing record with key: ${record.key});
            logger.info(Record value: ${JSON.stringify(record.value)});
            // You can add more processing logic here
        }
    },
    {
        value: {
            type: 'json',
        },
    }
);
```

Exemple .NET :

```
using Amazon.Lambda.Core;
using AWS.Lambda.Powertools.Kafka;
using AWS.Lambda.Powertools.Kafka.Json;
```

```
using AWS.Lambda.Powertools.Logging;
using Com.Example;

// Assembly attribute to enable the Lambda function's Kafka event to be converted
 into a .NET class.
[assembly: LambdaSerializer(typeof(PowertoolsKafkaJsonSerializer))]

namespace JsonClassLibrary;

public class Function
{
    public string FunctionHandler(ConsumerRecords<string, CustomerProfile> records,
 ILambdaContext context)
    {
        foreach (var record in records)
        {
            Logger.LogInformation("Processing messagem from topic: {topic}",
 record.Topic);
            Logger.LogInformation("Partition: {partition}, Offset: {offset}",
 record.Partition, record.Offset);
            Logger.LogInformation("Produced at: {timestamp}", record.Timestamp);

            foreach (var header in record.Headers.DecodedValues())
            {
                Logger.LogInformation($"{header.Key}: {header.Value}");
            }

            Logger.LogInformation("Processing order for: {fullName}",
 record.Value.FullName);
        }

        return "Processed " + records.Count() + " records";
    }
}
```

Méthodes d’authentification pour votre registre de schémas

Pour utiliser un registre de schémas, Lambda doit pouvoir y accéder de façon sécurisée. Si vous travaillez avec un registre de AWS Glue schémas, Lambda s'appuie sur l'authentification IAM. Cela signifie que le rôle d'exécution de votre fonction doit disposer des autorisations suivantes pour accéder au AWS Glue registre :

- GetRegistrydans la référence de l'API AWS Glue Web
- GetSchemaVersiondans la référence de l'API AWS Glue Web

Exemple de politique IAM requise :

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "glue:GetRegistry",
                "glue:GetSchemaVersion"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

> **Note**
>
> Pour les registres de AWS Glue schémas, si vous fournissez `AccessConfigs` un AWS Glue registre, Lambda renverra une exception de validation.

Si vous travaillez avec un registre de schémas Confluent, vous pouvez choisir l'une des trois méthodes d'authentification prises en charge pour le `Type` paramètre de votre KafkaSchemaRegistryAccessConfigobjet :

- BASIC_AUTH : Lambda utilise le nom d'utilisateur et le mot de passe ou l'authentification par clé d'API et secret d'API pour accéder à votre registre. Si vous choisissez cette option, indiquez l'ARN Secrets Manager contenant vos informations d'identification dans le champ URI.
- CLIENT_CERTIFICATE_TLS_AUTH : Lambda utilise l'authentification TLS mutuelle avec les certificats clients. Pour utiliser cette option, Lambda doit avoir accès à la fois au certificat et à la clé

privée. Indiquez l'ARN Secrets Manager contenant ces informations d'identification dans le champ URI.

- NO_AUTH : le certificat CA public doit être signé par une autorité de certification située dans le magasin d'approbations Lambda. Pour un certificat CA/autosigné privé, vous configurez le certificat CA racine du serveur. Pour utiliser cette option, omettez le paramètre `AccessConfigs`.

En outre, si Lambda a besoin d'accéder à un certificat CA privé pour vérifier le certificat TLS de votre registre de schémas, choisissez `SERVER_ROOT_CA_CERT` comme `Type` et fournissez l'ARN Secrets Manager au certificat dans le champ URI.

> **Note**
>
> Pour configurer l'option `SERVER_ROOT_CA_CERT` dans la console, fournissez l'ARN secret contenant le certificat dans le champ Chiffrement.

La configuration d'authentification de votre registre de schémas est distincte de toute authentification que vous avez configurée pour votre cluster Kafka. Vous devez configurer les deux séparément, même s'ils utilisent des méthodes d'authentification similaires.

Gestion des erreurs et résolution des problèmes liés au registre de schémas

Lorsque vous utilisez un registre de schémas avec votre source d'événements Amazon MSK, vous pouvez rencontrer diverses erreurs. Cette section fournit des conseils sur les problèmes courants et leur résolution.

Erreurs de configuration

Ces erreurs se produisent lors de la configuration de votre registre de schémas.

Mode provisionné requis

Message d'erreur : `SchemaRegistryConfig is only available for Provisioned Mode. To configure Schema Registry, please enable Provisioned Mode by specifying MinimumPollers in ProvisionedPollerConfig.`

Résolution : activez le mode provisionné pour votre mappage des sources d'événements en configurant le paramètre `MinimumPollers` dans `ProvisionedPollerConfig`.

URL de registre de schémas non valide

Message d'erreur : `Malformed SchemaRegistryURI provided. Please provide a valid URI or ARN. For example, https://schema-registry.example.com:8081 or arn:aws:glue:us-east-1:123456789012:registry/ExampleRegistry.`

Solution : Fournissez une URL HTTPS valide pour le registre des schémas Confluent ou un ARN valide pour le registre des AWS Glue schémas.

Format d'enregistrement d'événement non valide ou manquant

Message d'erreur : `EventRecordFormat is a required field for SchemaRegistryConfig. Please provide one of supported format types: SOURCE, JSON.`

Résolution : Spécifiez SOURCE ou JSON EventRecordFormat dans la configuration de votre registre de schéma.

Attributs de validation en double

Message d'erreur : `Duplicate KEY/VALUE Attribute in SchemaValidationConfigs. SchemaValidationConfigs must contain at most one KEY/VALUE Attribute.`

Résolution : Supprimez les attributs KEY ou VALUE dupliqués de votre SchemaValidationConfigs. Chaque type d'attribut ne peut apparaître qu'une seule fois.

Configuration de validation manquante

Message d'erreur : `SchemaValidationConfigs is a required field for SchemaRegistryConfig.`

Solution : Ajoutez SchemaValidationConfigs à votre configuration en spécifiant au moins un attribut de validation (KEY ou VALUE).

Erreurs d'accès et d'autorisation

Ces erreurs se produisent lorsque Lambda ne peut pas accéder au registre de schémas en raison de problèmes d'autorisation ou d'authentification.

AWS Glue Accès au registre du schéma refusé

Message d'erreur : `Cannot access Glue Schema with provided role. Please ensure the provided role can perform the GetRegistry and GetSchemaVersion Actions on your schema.`

Résolution : ajoutez les autorisations requises (`glue:GetRegistry` et `glue:GetSchemaVersion`) au rôle d'exécution de votre fonction.

Accès au registre de schémas Confluent refusé

Message d'erreur : `Cannot access Confluent Schema with the provided access configuration.`

Solution : vérifiez que vos informations d'authentification (stockées dans Secrets Manager) sont correctes et que vous disposez des autorisations nécessaires pour accéder au registre de schémas.

Registre des AWS Glue schémas entre comptes

Message d'erreur : `Cross-account Glue Schema Registry ARN not supported.`

Solution : utilisez un registre de AWS Glue schémas enregistré dans le même AWS compte que votre fonction Lambda.

Registre de AWS Glue schémas interrégional

Message d'erreur : `Cross-region Glue Schema Registry ARN not supported.`

Solution : utilisez un registre de AWS Glue schémas situé dans la même région que votre fonction Lambda.

Problèmes d'accès au secret

Message d'erreur : `Lambda received InvalidRequestException from Secrets Manager.`

Solution : Vérifiez que le rôle d'exécution de votre fonction est autorisé à accéder au secret et que celui-ci n'est pas chiffré avec une AWS KMS clé par défaut si vous y accédez depuis un autre compte.

Erreurs de connexion

Ces erreurs se produisent lorsque Lambda ne parvient pas à établir une connexion au registre de schémas.

Problèmes de connectivité VPC

- Message d'erreur : `Cannot connect to your Schema Registry. Your Kafka cluster's VPC must be able to connect to the schema registry. You can provide access by configuring AWS PrivateLink or a NAT Gateway or VPC Peering between Kafka Cluster VPC and the schema registry VPC.`
- Solution : configurez votre réseau VPC pour autoriser les connexions au registre de schéma à l'aide AWS PrivateLink d'une passerelle NAT ou d'un peering VPC.

Échec d'établissement d'une liaison TLS

- Message d'erreur : `Unable to establish TLS handshake with the schema registry. Please provide correct CA-certificate or client certificate using Secrets Manager to access your schema registry.`
- Solution : vérifiez que vos certificats CA et vos certificats clients (pour mTLS) sont corrects et bien configurés dans Secrets Manager.

étranglement

- Message d'erreur : `Receiving throttling errors when accessing the schema registry. Please increase API TPS limits for your schema registry.`
- Résolution : augmentez les limites de débit d'API pour votre registre de schémas ou réduisez le taux de demandes provenant de votre application.

Erreurs de registre de schémas autogéré

- Message d'erreur : `Lambda received an internal server an unexpected error from the provided self-managed schema registry.`
- Solution : vérifiez l'état et la configuration de votre serveur de registre de schémas autogéré.

Traitement à faible latence pour les sources d'événements Kafka

AWS Lambda prend en charge de manière native le traitement des événements à faible latence pour les applications qui nécessitent des latences constantes de bout en bout inférieures à 100 millisecondes. Cette page fournit des informations de configuration et des recommandations pour permettre des flux de travail à faible latence.

Activer le traitement à faible latence

Pour activer le traitement à faible latence sur un mappage des sources d'événements Kafka, la configuration de base suivante est requise :

- Activez le mode provisionné. Pour de plus amples informations, consultez Mode alloué.
- Définissez le paramètre du mappage des sources d'événements `MaximumBatchingWindowInSeconds` sur 0. Pour de plus amples informations, consultez Comportement de traitement par lots.

Peaufinage de votre ESM Kafka à faible latence

Tenez compte des recommandations suivantes pour optimiser votre mappage des sources d'événements Kafka pour une faible latence :

Configuration du mode provisionné

En mode provisionné pour le mappage des sources d'événements Kafka, Lambda vous permet d'optimiser le débit de votre mappage des sources d'événements en configurant des nombres minimal et maximal de ressources appelées sondeurs d'événements. Un sondeur d'événements (ou sondeur) représente une ressource de calcul qui sous-tend le mappage des sources d'événements en mode provisionné et alloue un débit allant jusqu'à 5 Mo/s. Chaque sondeur d'événements prend en charge jusqu'à 5 invocations Lambda simultanées.

Pour déterminer la configuration optimale de sondeurs pour votre application, tenez compte de votre taux d'ingestion maximal et de vos exigences de traitement. Étudions un exemple simplifié :

Avec une taille de lot de 20 enregistrements et une durée de fonction cible moyenne de 50 ms, chaque sondeur peut traiter 2 000 enregistrements par seconde dans la limite de 5 Mo/s. Le calcul est le suivant : (20 enregistrements × 1 000 ms/50 ms) × 5 appels Lambda simultanés. Par conséquent, si le taux d'ingestion maximal souhaité est de 20 000 enregistrements par seconde, vous aurez besoin d'au moins 10 sondeurs d'événements.

> **Note**
>
> Nous vous recommandons de prévoir des sondeurs d'événements supplémentaires en tant que tampon de sûreté afin d'éviter de fonctionner constamment à pleine capacité.

Le mode provisionné met à l'échelle vos sondeurs d'événements automatiquement en fonction des modèles de trafic dans les limites minimales et maximales de sondeurs d'événements configurées, ce qui peut déclencher un rééquilibrage, et par conséquent introduire une latence supplémentaire. Vous pouvez désactiver l'autoscaling en configurant la même valeur pour le nombre minimal et maximal de sondeurs d'événements.

Considérations supplémentaires

Voici quelques considérations supplémentaires :

- Les démarrages à froid à partir de l'invocation de votre fonction cible Lambda peuvent augmenter la latence de bout en bout. Pour réduire ce risque, pensez à activer la simultanéité provisionnée ou SnapStart sur la fonction cible de votre mappage des sources d'événements. En outre, optimisez l'allocation de mémoire de votre fonction pour garantir des exécutions cohérentes et optimales.
- Lorsque vous définissez `MaximumBatchingWindowInSeconds` sur 0, Lambda traite immédiatement tous les enregistrements disponibles sans attendre de remplir la taille complète du lot. Par exemple, si la taille de votre lot est définie sur 1 000 enregistrements, mais que seuls 100 enregistrements sont disponibles, Lambda traitera ces 100 enregistrements immédiatement plutôt que d'attendre que les 1 000 enregistrements complets soient accumulés.

> **Important**
>
> La configuration optimale pour un traitement à faible latence varie considérablement selon votre charge de travail spécifique. Nous vous recommandons vivement de tester différentes configurations avec votre charge de travail réelle afin de déterminer les meilleurs paramètres pour votre cas d'utilisation.

Configuration des contrôles de gestion des erreurs pour les sources d'événements Kafka

Vous pouvez configurer la façon dont Lambda gère les erreurs et les nouvelles tentatives pour vos mappages de sources d'événements Kafka. Ces configurations vous aident à contrôler la manière dont Lambda traite les enregistrements ayant échoué et gère le comportement des nouvelles tentatives.

Configurations de nouvelle tentative disponibles

Les configurations de nouvelle tentative suivantes sont disponibles pour Amazon MSK et les sources d'événements Kafka autogérées :

- Nombre maximal de tentatives : nombre maximal de tentatives de Lambda lorsque votre fonction renvoie une erreur. Cela ne compte pas la tentative d'invocation initiale. La valeur par défaut est -1 (infini).
- Age maximum d'enregistrement : âge maximum d'un enregistrement que Lambda envoie à votre fonction. La valeur par défaut est -1 (infini).
- Fractionner le lot en cas d'erreur : lorsque votre fonction renvoie une erreur, divisez le lot en deux lots plus petits et réessayez chacun séparément. Cela permet d'isoler les enregistrements problématiques.
- Réponse partielle par lot : autorisez votre fonction à renvoyer des informations sur les enregistrements d'un lot dont le traitement a échoué, afin que Lambda puisse réessayer uniquement les enregistrements ayant échoué.

Configuration des contrôles de gestion des erreurs (console)

Vous pouvez configurer le comportement des nouvelles tentatives lors de la création ou de la mise à jour d'un mappage de source d'événements Kafka dans la console Lambda.

Pour configurer le comportement des nouvelles tentatives pour une source d'événements Kafka (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom de votre fonction.
3. Effectuez l’une des actions suivantes :

- Pour ajouter un nouveau déclencheur Kafka, sous Vue d'ensemble des fonctions, choisissez Ajouter un déclencheur.
- Pour modifier un déclencheur Kafka existant, choisissez-le, puis choisissez Modifier.

4. Sous Configuration du sondeur d'événements, sélectionnez le mode provisionné pour configurer les contrôles de gestion des erreurs :

   a. Pour Réessayer, entrez le nombre maximum de tentatives (0 à 10 000, ou -1 pour un nombre infini).
   b. Pour Age maximum d'enregistrement, entrez l'âge maximum en secondes (60-604800, ou -1 pour un nombre infini).
   c. Pour activer le fractionnement par lots en cas d'erreur, sélectionnez Fractionner le lot en cas d'erreur.
   d. Pour activer la réponse partielle par lots, sélectionnez ReportBatchItemFailures.

5. Choisissez Ajouter ou Enregistrer.

Configuration du comportement des nouvelles tentatives ()AWS CLI

Utilisez les AWS CLI commandes suivantes pour configurer le comportement des nouvelles tentatives pour vos mappages de sources d'événements Kafka.

Création d'un mappage de sources d'événements avec des configurations de nouvelle tentative

L'exemple suivant crée un mappage de source d'événements Kafka autogéré avec des contrôles de gestion des erreurs :

```
aws lambda create-event-source-mapping \
  --function-name my-kafka-function \
  --topics my-kafka-topic \
  --source-access-configuration Type=SASL_SCRAM_512_AUTH,URI=arn:aws:secretsmanager:us-
east-1:111122223333:secret:MyBrokerSecretName \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":
["abc.xyz.com:9092"]}}' \
  --starting-position LATEST \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=1 \
  --maximum-retry-attempts 3 \
  --maximum-record-age-in-seconds 3600 \
  --bisect-batch-on-function-error \
  --function-response-types "ReportBatchItemFailures"
```

Pour les sources d'événements Amazon MSK :

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-fd1b-45ad-85dd-15b4a5a6247e-2 \
  --topics AWSMSKKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function \
  --source-access-configurations '[{"Type": "SASL_SCRAM_512_AUTH","URI": "arn:aws:secretsmanager:us-east-1:111122223333:secret:my-secret"}]' \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=1 \
  --maximum-retry-attempts 3 \
  --maximum-record-age-in-seconds 3600 \
  --bisect-batch-on-function-error \
  --function-response-types "ReportBatchItemFailures"
```

Mise à jour des configurations de nouvelle tentative

Utilisez la `update-event-source-mapping` commande pour modifier les configurations de nouvelle tentative pour un mappage de source d'événement existant :

```
aws lambda update-event-source-mapping \
  --uuid 12345678-1234-1234-1234-123456789012 \
  --maximum-retry-attempts 5 \
  --maximum-record-age-in-seconds 7200 \
  --bisect-batch-on-function-error \
  --function-response-types "ReportBatchItemFailures"
```

PartialBatchResponse

La réponse partielle par lots, également connue sous le nom de ReportBatchItemFailures, est une fonctionnalité clé pour la gestion des erreurs dans l'intégration de Lambda aux sources Kafka. Sans cette fonctionnalité, lorsqu'une erreur survient dans l'un des éléments d'un lot, cela entraîne le retraitement de tous les messages de ce lot. Lorsque la réponse par lots partielle est activée et implémentée, le gestionnaire renvoie des identifiants uniquement pour les messages ayant échoué, ce qui permet à Lambda de réessayer uniquement ces éléments spécifiques. Cela permet de mieux contrôler le traitement des lots contenant des messages d'échec.

Pour signaler des erreurs par lots, vous allez utiliser le schéma JSON suivant :

```
{
```

```
  "batchItemFailures": [
    {
      "itemIdentifier": {
        "topic-partition": "topic-partition_number",
        "offset": 100
      }
    },
    ...
  ]
}
```

> **Important**
>
> Si vous renvoyez un fichier JSON valide vide ou nul, le mappage de la source d'événements considérera qu'un lot a été traité avec succès. Tout topic-partition_number ou offset non valide renvoyé qui n'était pas présent dans l'événement invoqué sera considéré comme un échec et le lot entier sera réessayé.

Les exemples de code suivants montrent comment implémenter une réponse par lots partielle pour les fonctions Lambda qui reçoivent des événements provenant de sources Kafka. La fonction signale les défaillances échecs d'articles par lots dans la réponse, en indiquant à Lambda de réessayer ces messages ultérieurement.

Voici une implémentation du gestionnaire Lambda en Python qui illustre cette approche :

```
import base64
from typing import Any, Dict, List

def lambda_handler(event: Dict[str, Any], context: Any) -> Dict[str, List[Dict[str,
 Dict[str, Any]]]]:
    failures: List[Dict[str, Dict[str, Any]]] = []
    records_dict = event.get("records", {})

    for topic_partition, records_list in records_dict.items():
        for record in records_list:
            topic = record.get("topic")
            partition = record.get("partition")
            offset = record.get("offset")
            value_b64 = record.get("value")
```

```
            try:
                data = base64.b64decode(value_b64).decode("utf-8")
                process_message(data)
            except Exception as exc:
                print(f"Failed to process record topic={topic} partition={partition}
 offset={offset}: {exc}")
                item_identifier: Dict[str, Any] = {
                    "topic-partition": f"{topic}-{partition}",
                    "offset": int(offset) if offset is not None else None,
                }
                failures.append({"itemIdentifier": item_identifier})

    return {"batchItemFailures": failures}

def process_message(data: str) -> None:
    # Your business logic for a single message
    pass
```

Voici une version de Node.js :

```
const { Buffer } = require("buffer");

const handler = async (event) => {
  const failures = [];

  for (let topicPartition in event.records) {
    const records = event.records[topicPartition];

    for (const record of records) {
      const topic = record.topic;
      const partition = record.partition;
      const offset = record.offset;
      const valueBase64 = record.value;
      const data = Buffer.from(valueBase64, "base64").toString("utf8");

      try {
        await processMessage(data);
      } catch (error) {
        console.error("Failed to process record", { topic, partition, offset, error });
        const itemIdentifier = {
          "topic-partition": `${topic}-${partition}`,
          offset: Number(offset),
        };
```

```
        failures.push({ itemIdentifier });
      }
    }
  }

  return { batchItemFailures: failures };
};

async function processMessage(payload) {
  // Your business logic for a single message
}

module.exports = { handler };
```

Capture de lots supprimés pour des sources d'événement Amazon MSK et Apache Kafka autogéré

Pour retenir les enregistrements des invocations de mappage de sources d'événements qui ont échoué, ajoutez une destination au mappage des sources d'événements de votre fonction. Chaque enregistrement envoyé à la destination est un document JSON contenant les métadonnées sur l'invocation ayant échoué. Pour les destinations Amazon S3, Lambda envoie également l'intégralité de l'enregistrement d'invocation avec les métadonnées. Vous pouvez configurer n'importe quelle rubrique Amazon SNS, n'importe quelle file d'attente Amazon SQS, n'importe quel compartiment Amazon S3 ou Kafka comme destination.

Avec les destinations Amazon S3, vous pouvez utiliser la fonctionnalité Notifications d'événements Amazon S3 pour recevoir des notifications lorsque des objets sont chargés dans votre compartiment S3 de destination. Vous pouvez également configurer les notifications d'événements S3 pour invoquer une autre fonction Lambda afin d'effectuer un traitement automatique des lots ayant échoué.

Votre rôle d'exécution doit disposer d'autorisations pour la destination :

- Pour une destination SQS : sqs : SendMessage
- Pour une destination SNS : SNS:Publish
- Pour une destination S3 : s3 : PutObject et s3 : ListBucket
- Pour une destination Kafka : kafka-cluster : WriteData

Vous pouvez configurer un sujet Kafka comme destination en cas d'échec pour vos mappages de sources d'événements Kafka. Lorsque Lambda ne parvient pas à traiter les enregistrements après avoir épuisé toutes les tentatives ou lorsque les enregistrements dépassent l'âge maximum, Lambda envoie les enregistrements ayant échoué à la rubrique Kafka spécifiée pour un traitement ultérieur. Reportez-vous à the section called "Destination de Kafka en cas d'échec".

Vous devez déployer un point de terminaison de VPC pour votre service de destination en cas de panne au sein de votre VPC de cluster Kafka.

En outre, si vous avez configuré une clé KMS sur votre destination, Lambda a besoin des autorisations suivantes en fonction du type de destination :

- Si vous avez activé le chiffrement avec votre propre clé KMS pour une destination S3, kms : GenerateDataKey est obligatoire. Si la clé KMS et la destination du compartiment S3 se trouvent

dans un compte différent de celui de votre fonction Lambda et de votre rôle d'exécution, configurez la clé KMS pour qu'elle approuve le rôle d'exécution à autoriser. kms: GenerateDataKey

- Si vous avez activé le chiffrement avec votre propre clé KMS pour la destination SQS, KMS:Decrypt et kms : sont obligatoires. GenerateDataKey Si la clé KMS et la destination de la file d'attente SQS se trouvent dans un compte différent de celui de votre fonction Lambda et de votre rôle d'exécution, configurez la clé KMS pour qu'elle approuve le rôle d'exécution afin d' kms:autoriser Decrypt kms: GenerateDataKey, kms DescribeKey : et kms :. ReEncrypt
- Si vous avez activé le chiffrement avec votre propre clé KMS pour la destination SNS, KMS:Decrypt et kms : sont requis. GenerateDataKey Si la clé KMS et la destination de la rubrique SNS se trouvent dans un compte différent de celui de votre fonction Lambda et de votre rôle d'exécution, configurez la clé KMS pour qu'elle approuve le rôle d'exécution afin d' kms:autoriser Decrypt kms: GenerateDataKey, kms DescribeKey : et kms :. ReEncrypt

Configuration des destinations en cas d’échec pour un mappage des sources d’événements Kafka

Pour configurer une destination en cas de panne à l’aide de la console, procédez comme suit :

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Sous Function overview (Vue d’ensemble de la fonction), choisissez Add destination (Ajouter une destination).
4. Pour Source, choisissez Invocation du mappage des sources d’événements.
5. Pour le mappage des sources d’événements, choisissez une source d’événements configurée pour cette fonction.
6. Pour Condition, sélectionnez En cas d’échec. Pour les invocations de mappage des sources d’événements, il s’agit de la seule condition acceptée.
7. Pour Type de destination, choisissez le type de destination auquel Lambda envoie les enregistrements d’invocation.
8. Pour Destination, choisissez une ressource.
9. Choisissez Enregistrer.

Vous pouvez également configurer une destination en cas de panne à l'aide de l' AWS CLI. Par exemple, la create-event-source-mappingcommande suivante ajoute un mappage de source d'événement avec une destination SQS en cas de défaillance pour : `MyFunction`

```
aws lambda create-event-source-mapping \
--function-name "MyFunction" \
--event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-
east-1:123456789012:dest-queue"}}'
```

La update-event-source-mappingcommande suivante ajoute une destination S3 en cas de défaillance à la source d'événements associée à l'entrée `uuid` :

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": "arn:aws:s3:::dest-bucket"}}'
```

Pour supprimer une destination, entrez une chaîne vide comme argument du paramètre `destination-config` :

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

Pratiques exemplaires en matière de sécurité pour les destinations Amazon S3

La suppression d'un compartiment S3 configuré comme destination sans supprimer la destination de la configuration de votre fonction peut engendrer un risque de sécurité. Si un autre utilisateur connaît le nom de votre compartiment de destination, il peut recréer le compartiment dans son Compte AWS. Les enregistrements des invocations ayant échoué seront envoyés dans son compartiment, exposant potentiellement les données de votre fonction.

> **Warning**
>
> Pour vous assurer que les enregistrements d'invocation de votre fonction ne peuvent pas être envoyés vers un compartiment S3 d'un autre Compte AWS, ajoutez une condition au rôle d'exécution de votre fonction qui limite `s3:PutObject` les autorisations aux compartiments de votre compte.

L'exemple suivant présente une politique IAM qui limite les autorisations `s3:PutObject` de votre fonction aux seuls compartiments de votre compte. Cette politique donne également à Lambda l'autorisation `s3:ListBucket` dont il a besoin pour utiliser un compartiment S3 comme destination.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "S3BucketResourceAccountWrite",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::*/*",
                "arn:aws:s3:::*"
            ],
            "Condition": {
                "StringEquals": {
                    "s3:ResourceAccount": "111122223333"
                }
            }
        }
    ]
}
```

Pour ajouter une politique d'autorisations au rôle d'exécution de votre fonction à l'aide du AWS Management Console ou AWS CLI, reportez-vous aux instructions des procédures suivantes :

Console

Pour ajouter une politique d'autorisations au rôle d'exécution d'une fonction (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction Lambda dont vous voulez modifier le rôle d'exécution.
3. Sous l'onglet Configuration, sélectionnez Autorisations.
4. Sous l'onglet Rôle d'exécution, sélectionnez le Nom du rôle de votre fonction pour ouvrir la page de console IAM du rôle.
5. Ajoutez une politique d'autorisations de au rôle en procédant comme suit :

a. Dans le volet Politiques d'autorisations, choisissez Ajouter des autorisations, puis Créer une politique en ligne.

b. Dans l'Éditeur de politique, sélectionnez JSON.

c. Collez la politique que vous souhaitez ajouter dans l'éditeur (en remplacement du JSON existant), puis choisissez Suivant.

d. Sous Détails de la politique, saisissez un Nom de la politique.

e. Choisissez Create Policy (Créer une politique).

AWS CLI

Pour ajouter une politique d'autorisations au rôle d'exécution d'une fonction (CLI)

1. Créez un document de politique JSON avec les autorisations requises et enregistrez-le dans un répertoire local.
2. Utilisez la commande `put-role-policy` de la CLI IAM pour ajouter des autorisations au rôle d'exécution de votre fonction. Exécutez la commande suivante depuis le répertoire dans lequel vous avez enregistré votre document de politique JSON et remplacez le nom du rôle, le nom de la politique et le document de politique par vos propres valeurs.

```
aws iam put-role-policy \
--role-name my_lambda_role \
--policy-name LambdaS3DestinationPolicy \
--policy-document file://my_policy.json
```

Exemple d'enregistrement d'invocation SNS et SQS

L'exemple suivant montre ce que Lambda envoie à une rubrique SNS ou à une destination de file d'attente SQS pour une invocation de source d'événement Kafka qui a échoué. Chacune des clés sous `recordsInfo` contient à la fois le sujet et la partition Kafka, séparés par un trait d'union. Par exemple, pour la clé`"Topic-0"`, `Topic` est la rubrique Kafka, et `0` est la partition. Pour chaque sujet et chaque partition, vous pouvez utiliser les décalages et les données d'horodatage pour trouver les enregistrements d'invocation d'origine.

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
```

```
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted" | "MaximumPayloadSizeExceeded",
        "approximateInvokeCount": 1
    },
    "responseContext": { // null if record is MaximumPayloadSizeExceeded
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KafkaBatchInfo": {
        "batchSize": 500,
        "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
        "bootstrapServers": "...",
        "payloadSize": 2039086, // In bytes
        "recordsInfo": {
            "Topic-0": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            },
            "Topic-1": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            }
        }
    }
}
```

Exemple d’enregistrement d’invocation de destination S3

Pour les destinations S3, Lambda envoie l’intégralité de l’enregistrement d’invocation ainsi que les métadonnées à la destination. L’exemple suivant montre que Lambda envoie vers une destination de compartiment S3 en cas d’échec d’une invocation de source d’événement Kafka. Outre tous les

champs de l'exemple précédent pour les destinations SQS et SNS, le champ `payload` contient l'enregistrement d'invocation d'origine sous forme de chaîne JSON échappée.

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted" | "MaximumPayloadSizeExceeded",
        "approximateInvokeCount": 1
    },
    "responseContext": { // null if record is MaximumPayloadSizeExceeded
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KafkaBatchInfo": {
        "batchSize": 500,
        "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
        "bootstrapServers": "...",
        "payloadSize": 2039086, // In bytes
        "recordsInfo": {
            "Topic-0": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            },
            "Topic-1": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            }
        }
    },
    "payload": "<Whole Event>" // Only available in S3
```

```
}
```

> **Tip**
>
> Nous vous recommandons d'activer la gestion des versions S3 sur votre compartiment de destination.

Utiliser un sujet Kafka comme destination en cas d'échec

Vous pouvez configurer un sujet Kafka comme destination en cas d'échec pour vos mappages de sources d'événements Kafka. Lorsque Lambda ne parvient pas à traiter les enregistrements après avoir épuisé toutes les tentatives ou lorsque les enregistrements dépassent l'âge maximum, Lambda envoie les enregistrements ayant échoué à la rubrique Kafka spécifiée pour un traitement ultérieur.

Comment fonctionne une destination Kafka en cas de défaillance

Lorsque vous configurez un sujet Kafka comme destination en cas d'échec, Lambda agit en tant que producteur Kafka et écrit les enregistrements ayant échoué dans le sujet de destination. Cela crée un modèle de sujet mort (DLT) au sein de votre infrastructure Kafka.

- Même exigence de cluster — Le sujet de destination doit exister dans le même cluster Kafka que vos sujets source.
- Contenu réel des enregistrements : les destinations Kafka reçoivent les enregistrements défaillants réels ainsi que les métadonnées relatives aux échecs.
- Prévention de la récursivité : Lambda empêche les boucles infinies en bloquant les configurations dans lesquelles les sujets source et destination sont identiques.

Configuration d'une destination Kafka en cas de panne

Vous pouvez configurer un sujet Kafka comme destination en cas d'échec lors de la création ou de la mise à jour d'un mappage de source d'événements Kafka.

Configuration d'une destination Kafka (console)

Pour configurer un sujet Kafka comme destination en cas d'échec (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom de votre fonction.
3. Effectuez l’une des actions suivantes :
   - Pour ajouter un nouveau déclencheur Kafka, sous Vue d'ensemble des fonctions, choisissez Ajouter un déclencheur.
   - Pour modifier un déclencheur Kafka existant, choisissez-le, puis choisissez Modifier.
4. Sous Paramètres supplémentaires, pour Destination en cas d'échec, choisissez le sujet Kafka.

5. Dans le champ Nom du sujet, entrez le nom du sujet Kafka auquel vous souhaitez envoyer les enregistrements ayant échoué.
6. Choisissez Ajouter ou Enregistrer.

Configuration d'une destination Kafka ()AWS CLI

Utilisez le `kafka://` préfixe pour spécifier un sujet Kafka comme destination en cas d'échec.

Création d'un mappage des sources d'événements avec la destination Kafka

L'exemple suivant crée un mappage de source d'événement Amazon MSK avec un sujet Kafka comme destination en cas d'échec :

```
aws lambda create-event-source-mapping \
  --function-name my-kafka-function \
  --topics AWSKafkaTopic \
  --event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/my-cluster/abc123 \
  --starting-position LATEST \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=3 \
  --destination-config '{"OnFailure":{"Destination":"kafka://failed-records-topic"}}'
```

Pour Kafka autogéré, utilisez la même syntaxe :

```
aws lambda create-event-source-mapping \
  --function-name my-kafka-function \
  --topics AWSKafkaTopic \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":
["abc.xyz.com:9092"]}}' \
  --starting-position LATEST \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=3 \
  --destination-config '{"OnFailure":{"Destination":"kafka://failed-records-topic"}}'
```

Mettre à jour une destination Kafka

Utilisez la `update-event-source-mapping` commande pour ajouter ou modifier une destination Kafka :

```
aws lambda update-event-source-mapping \
  --uuid 12345678-1234-1234-1234-123456789012 \
  --destination-config '{"OnFailure":{"Destination":"kafka://failed-records-topic"}}'
```

Format d'enregistrement pour une destination Kafka

Lorsque Lambda envoie des enregistrements ayant échoué à un sujet Kafka, chaque message contient à la fois des métadonnées relatives à l'échec et le contenu réel de l'enregistrement.

Métadonnées relatives aux défaillances

Les métadonnées incluent des informations sur les raisons de l'échec de l'enregistrement et des détails sur le lot d'origine :


```
{
  "requestContext": {
    "requestId": "e4b46cbf-b738-xmpl-8880-a18cdf61200e",
    "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function:
$LATEST",
    "condition": "RetriesExhausted",
    "approximateInvokeCount": 3
  },
  "responseContext": {
    "statusCode": 200,
    "executedVersion": "$LATEST",
    "functionError": "Unhandled"
  },
  "version": "1.0",
  "timestamp": "2019-11-14T18:16:05.568Z",
  "KafkaBatchInfo": {
    "batchSize": 1,
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-cluster/abc123",
    "bootstrapServers": "b-1.mycluster.abc123.kafka.us-east-1.amazonaws.com:9098",
    "payloadSize": 1162,
    "recordInfo": {
      "offset": "49601189658422359378836298521827638475320189012309704722",
      "timestamp": "2019-11-14T18:16:04.835Z"
    }
  },
  "payload": {
    "bootstrapServers": "b-1.mycluster.abc123.kafka.us-east-1.amazonaws.com:9098",
    "eventSource": "aws:kafka",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-cluster/abc123",
    "records": {
      "my-topic-0": [
        {
          "headers": [],
```

```
          "key": "dGVzdC1rZXk=",
          "offset": 100,
          "partition": 0,
          "timestamp": 1749116692330,
          "timestampType": "CREATE_TIME",
          "topic": "my-topic",
          "value": "dGVzdC12YWx1ZQ=="
        }
      ]
    }
  }
}
```

Comportement des clés de partition

Lambda utilise la même clé de partition que celle de l'enregistrement d'origine lors de la production vers le sujet de destination. Si l'enregistrement d'origine ne contenait aucune clé, Lambda utilise le partitionnement circulaire par défaut de Kafka entre toutes les partitions disponibles dans le sujet de destination.

Exigences et limitations

- Mode provisionné requis — Une destination Kafka en cas de défaillance n'est disponible que pour les mappages de sources d'événements avec le mode provisionné activé.
- Même cluster uniquement — Le sujet de destination doit exister dans le même cluster Kafka que vos sujets source.
- Autorisations de sujet : le mappage de la source de votre événement doit disposer d'autorisations d'écriture sur le sujet de destination. Exemple :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ClusterPermissions",
            "Effect": "Allow",
            "Action": [
                "kafka-cluster:Connect",
                "kafka-cluster:DescribeCluster",
                "kafka-cluster:DescribeTopic",
                "kafka-cluster:WriteData",
                "kafka-cluster:ReadData"
```

```
            ],
            "Resource": [
                "arn:aws:kafka:*:*:cluster/*"
            ]
        },
        {
            "Sid": "TopicPermissions",
            "Effect": "Allow",
            "Action": [
                "kafka-cluster:DescribeTopic",
                "kafka-cluster:WriteData",
                "kafka-cluster:ReadData"
            ],
            "Resource": [
                "arn:aws:kafka:*:*:topic/*/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "kafka:DescribeCluster",
                "kafka:GetBootstrapBrokers",
                "kafka:Produce"
            ],
            "Resource": "arn:aws:kafka:*:*:cluster/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:CreateNetworkInterface",
                "ec2:DescribeNetworkInterfaces",
                "ec2:DeleteNetworkInterface",
                "ec2:DescribeSubnets",
                "ec2:DescribeSecurityGroups"
            ],
            "Resource": "*"
        }
    ]
}
```

- Aucune récursivité : le nom du sujet de destination ne peut pas être le même que celui de votre sujet source.

Résolution des erreurs de mappage des sources d'événements Kafka

Les rubriques suivantes fournissent des conseils de dépannage pour les erreurs et les problèmes que vous pouvez rencontrer en utilisant Amazon MSK ou Apache Kafka autogéré avec Lambda.

Pour obtenir de l'aide supplémentaire en matière de résolution des problèmes, consultez le Centre de connaissances AWS.

Authentification et erreurs d'autorisation

Si l'une des autorisations requises pour consommer des données du cluster Kafka est manquante, Lambda affiche l'un des messages d'erreur suivants dans le mappage des sources d'événements sous LastProcessingResult.

Messages d'erreur

- Le cluster n'a pas pu autoriser Lambda
- Échec de l'authentification SASL
- Le serveur n'a pas réussi à authentifier Lambda
- Lambda n'a pas réussi à authentifier le serveur
- Le certificat ou la clé privée fournis n'est pas valide

Le cluster n'a pas pu autoriser Lambda

Pour SASL/SCRAM ou mTLS, cette erreur indique que l'utilisateur fourni ne dispose pas de toutes les autorisations de liste de contrôle d'accès (ACL) Kafka requises suivantes :

- DescripbeConfigs Cluster
- Décrire un groupe
- Lire le groupe
- Décrire la rubrique
- Lire la rubrique

Lorsque vous créez des listes ACL Kafka avec les autorisations `kafka-cluster` requises, vous devez spécifier la rubrique et le groupe en tant que ressources. Le nom de la rubrique doit correspondre à la rubrique dans le mappage des sources d'événements. Le nom du groupe doit correspondre à l'UUID du mappage des sources d'événements.

Une fois que vous avez ajouté les autorisations requises au rôle d'exécution, il peut y avoir un délai de plusieurs minutes avant que les modifications ne prennent effet.

Échec de l'authentification SASL

Pour SASL/SCRAM ou SASL/PLAIN, cette erreur indique que les informations d'identification fournies ne sont pas valides.

Pour le contrôle d'accès IAM, le rôle d'exécution ne dispose pas des autorisations `kafka-cluster:Connect` pour le cluster. Ajoutez cette autorisation au rôle et spécifiez l'Amazon Resource Name (ARN) du cluster en tant que ressource.

Cette erreur peut se produire de façon intermittente. Le cluster rejette les connexions une fois que le nombre de connexions TCP dépasse le Quota de service. Lambda fait marche arrière et tente de nouveau jusqu'à ce qu'une connexion soit réussie. Une fois que Lambda se connecte au cluster et interroge les enregistrements, le dernier résultat de traitement passe à `OK`.

Le serveur n'a pas réussi à authentifier Lambda

Cette erreur indique que l'agent Kafka n'a pas réussi à authentifier Lambda. Cette erreur se produit dans les conditions suivantes :

- Vous n'avez pas fourni de certificat client pour l'authentification mTLS.
- Vous avez fourni un certificat client, mais les courtiers ne sont pas configurés pour utiliser l'authentification mTLS.
- Les courtiers Kafka ne font pas confiance à un certificat client.

Lambda n'a pas réussi à authentifier le serveur

Cette erreur indique que Lambda n'a pas réussi à authentifier l'agent Kafka. Cette erreur se produit dans les conditions suivantes :

- Pour Apache Kafka autogéré : les agents Kafka utilisent des certificats autosignés ou une autorité de certification privée, mais n'ont pas fourni le certificat CA racine du serveur.
- Pour Apache Kafka autogéré : le certificat CA racine du serveur ne correspond pas à l'autorité de certification racine qui a signé le certificat de l'agent.
- La validation du nom d'hôte a échoué, car le certificat du courtier ne contient pas le nom DNS ou l'adresse IP du courtier comme autre nom de sujet.

Le certificat ou la clé privée fournis n'est pas valide

Cette erreur indique que le consommateur Kafka n'a pas pu utiliser le certificat ou la clé privée fournis. Assurez-vous que le certificat et la clé utilisent le format PEM et que le chiffrement de la clé privée utilise un algorithme PBES1.

API de mappage des sources d'événements

Lorsque vous ajoutez votre cluster Apache Kafka en tant que source d'événement pour votre fonction Lambda, si votre fonction rencontre une erreur, votre consommateur Kafka cesse de traiter les registres. Les consommateurs d'une partition de rubrique sont ceux qui s'abonnent à vos registres et qui les lisent et traitent. Vos autres consommateurs Kafka peuvent continuer à traiter les registres, à condition qu'ils ne rencontrent pas la même erreur.

Afin d'identifier les raisons pour lesquelles un consommateur cesse son traitement, vérifiez le champ `StateTransitionReason` dans la réponse de `EventSourceMapping`. La liste suivante décrit les erreurs de source d'événement que vous pouvez recevoir :

`ESM_CONFIG_NOT_VALID`

La configuration de mappage de source d'événement n'est pas valide.

`EVENT_SOURCE_AUTHN_ERROR`

Lambda n'a pas pu authentifier la source d'événement.

`EVENT_SOURCE_AUTHZ_ERROR`

Lambda ne dispose pas des autorisations requises pour accéder à la source d'événement.

`FUNCTION_CONFIG_NOT_VALID`

La configuration de la fonction n'est pas valide.

> **Note**
>
> Si vos registres d'événement Lambda dépassent la limite de taille autorisée de 6 Mo, il se peut qu'ils ne soient pas traités.

Invocation d'une fonction Lambda à l'aide d'un point de terminaison Amazon API Gateway

Vous pouvez créer une API web avec un point de terminaison HTTP pour votre fonction Lambda à l'aide d'Amazon API Gateway. API Gateway fournit des outils pour créer et documenter des sites Web APIs qui acheminent les requêtes HTTP vers les fonctions Lambda. Vous pouvez sécuriser l'accès à votre API avec des contrôles d'authentification et d'autorisation. APIs Vous pouvez gérer le trafic sur Internet ou n'être accessible qu'au sein de votre VPC.

> **Tip**
>
> Lambda propose deux méthodes pour appeler votre fonction via un point de terminaison HTTP : API Gateway et fonction Lambda. URLs Si vous ne savez pas quelle est la meilleure méthode pour votre cas d'utilisation, consultez the section called "API Gateway et fonction URLs".

Les ressources de votre API définissent une ou plusieurs méthodes, telles que GET ou POST. Les méthodes ont une intégration qui achemine les requêtes vers une fonction Lambda ou un autre type d'intégration. Vous pouvez définir chaque ressource et méthode individuellement, ou utiliser des types de ressource et de méthode spéciaux pour correspondre à toutes les demandes adaptées à un modèle. Une ressource proxy attrape tous les chemins sous une ressource. La méthode ANY attrape toutes les méthodes HTTP.

Sections

- Choix d'un type d'API
- Ajout d'un point de terminaison public à votre fonction Lambda
- Intégration de proxy
- Format des événements
- Format de la réponse
- Permissions
- Exemple d'application
- Le gestionnaire d'événements de Powertools pour Lambda AWS
- Didacticiel : Utiliser Lambda avec Amazon API Gateway
- Gestion des erreurs Lambda avec une API Gateway

- Sélection d'une méthode pour invoquer votre fonction Lambda à l'aide d'une requête HTTP

Choix d'un type d'API

API Gateway prend en charge trois types de fonctions APIs qui invoquent des fonctions Lambda :

- API HTTP : API légère à faible latence RESTful .
- API REST : API personnalisable et riche en fonctionnalités RESTful .
- WebSocket API : API Web qui maintient des connexions persistantes avec les clients pour une communication en duplex intégral.

HTTP APIs et REST APIs traitent RESTful APIs les requêtes HTTP et renvoient des réponses. APIs Les protocoles HTTP sont plus récents et sont créés avec l'API API Gateway version 2. Les fonctionnalités suivantes sont nouvelles pour le protocole HTTP APIs :

Fonctionnalités de l'API HTTP

- Déploiements automatiques – Lorsque vous modifiez des routages ou des intégrations, les modifications se déploient automatiquement sur des étapes pour lesquelles le déploiement automatique est activé.
- Étape par défaut – Vous pouvez créer une étape par défaut (`$default`) pour servir les demandes au chemin d'accès racine de l'URL de votre API. Pour les étapes nommées, vous devez inclure le nom de l'étape au début du chemin d'accès.
- Configuration CORS – Vous pouvez configurer votre API pour ajouter des en-têtes CORS aux réponses sortantes, au lieu de les ajouter manuellement dans votre code de fonction.

APIs Les REST sont RESTful APIs les classiques pris en charge par API Gateway depuis son lancement. REST propose APIs actuellement davantage de fonctionnalités de personnalisation, d'intégration et de gestion.

Fonctionnalités de l'API REST

- Types d'intégration : REST APIs prend en charge les intégrations Lambda personnalisées. Avec une intégration personnalisée, vous pouvez envoyer uniquement le corps de la requête à la fonction, ou appliquer un modèle de transformation au corps de requête avant de l'envoyer à la fonction.

- Contrôle d'accès : REST APIs prend en charge davantage d'options d'authentification et d'autorisation.
- Surveillance et suivi : REST APIs prend en charge le AWS X-Ray suivi et des options de journalisation supplémentaires.

Pour une comparaison détaillée, consultez Choisir entre HTTP APIs et REST APIs dans le guide du développeur d'API Gateway.

WebSocket APIs utilise également l'API API Gateway version 2 et prend en charge un ensemble de fonctionnalités similaires. Utilisez une WebSocket API pour les applications qui bénéficient d'une connexion permanente entre le client et l'API. WebSocket APIs fournissent une communication en duplex intégral, ce qui signifie que le client et l'API peuvent envoyer des messages en continu sans attendre de réponse.

Le protocole HTTP APIs prend en charge un format d'événement simplifié (version 2.0). Pour un exemple d'événement provenant d'une API HTTP, voir Créer des intégrations de AWS Lambda proxy pour HTTP APIs dans API Gateway.

Pour plus d'informations, consultez la section Création d'intégrations de AWS Lambda proxy pour HTTP APIs dans API Gateway.

Ajout d’un point de terminaison public à votre fonction Lambda

Ajouter un point de terminaison public à votre fonction Lambda

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Sous Function overview (Vue d’ensemble de la fonction), choisissez Add trigger (Ajouter un déclencheur).
4. Sélectionnez API Gateway.
5. Choisissez Créer une API ou Utiliser une API existante.
    a. Nouvelle API : pour Type d’API, choisissez API HTTP. Pour de plus amples informations, veuillez consulter Choix d’un type d’API.
    b. API existante : Sélectionnez l’API dans la liste déroulante ou entrez l’ID de l’API (par exemple, r3pmxmplak).
6. Pour Sécurité, choisissez Ouvrir.

7. Choisissez Ajouter.

Intégration de proxy

API Gateway comprend APIs des étapes, des ressources, des méthodes et des intégrations. L'étape et la ressource déterminent le chemin du point de terminaison :

Format du chemin d'accès de l'API

- `/prod/` – Etape `prod` et ressource racine.
- `/prod/user` – Etape `prod` et ressource `user`.
- `/dev/{proxy+}` – Routage quelconque à l'étape `dev`.
- `/` — (HTTP APIs) Le stage par défaut et la ressource racine.

Une intégration Lambda mappe une combinaison de chemin d'accès et de méthode HTTP à une fonction Lambda. Vous pouvez configurer API Gateway pour transmettre le corps de la demande HTTP tel quel (intégration personnalisée) ou pour encapsuler le corps de requête dans un document qui inclut toutes les informations de la demande, y compris les en-têtes, la ressource, le chemin et la méthode.

Pour plus d'informations, consultez Lambda proxy integrations in API Gateway.

Format des événements

Amazon API Gateway appelle votre fonction de manière synchrone avec un événement contenant une représentation JSON de la requête HTTP. Pour une intégration personnalisée, l'événement est le corps de la requête. Pour une intégration par proxy, l'événement a une structure définie. Pour obtenir un exemple d'événement de proxy à partir d'une API REST API Gateway, consultez la rubrique Input format of a Lambda function for proxy integration du guide du développeur API Gateway.

Format de la réponse

API Gateway attend une réponse de votre fonction et relaie le résultat à l'appelant. Pour une intégration personnalisée, vous définissez une réponse d'intégration et une réponse de méthode pour convertir la sortie de la fonction en réponse HTTP. Pour une intégration par proxy, la fonction doit répondre avec une représentation de la réponse dans un format spécifique.

L'exemple suivant montre un objet de réponse d'une fonction Node.js. L'objet de réponse représente une réponse HTTP réussie qui contient un document JSON.

Example index.js : objet de réponse d'intégration de proxy (Node.js)

```
var response = {
      "statusCode": 200,
      "headers": {
        "Content-Type": "application/json"
      },
      "isBase64Encoded": false,
      "multiValueHeaders": {
        "X-Custom-Header": ["My value", "My other value"],
      },
      "body": "{\n  \"TotalCodeSize\": 104330022,\n  \"FunctionCount\": 26\n}"
    }
```

L'environnement d'exécution Lambda sérialise l'objet réponse dans JSON et l'envoie à l'API. L'API analyse la réponse et l'utilise pour créer une réponse HTTP, qu'elle envoie ensuite au client qui a fait la demande d'origine.

Example Réponse HTTP

```
< HTTP/1.1 200 OK
  < Content-Type: application/json
  < Content-Length: 55
  < Connection: keep-alive
  < x-amzn-RequestId: 32998fea-xmpl-4268-8c72-16138d629356
  < X-Custom-Header: My value
  < X-Custom-Header: My other value
  < X-Amzn-Trace-Id: Root=1-5e6aa925-ccecxmplbae116148e52f036
  <
  {
    "TotalCodeSize": 104330022,
    "FunctionCount": 26
  }
```

Permissions

Amazon API Gateway obtient l'autorisation d'appeler votre fonction à partir de la politique basée sur une ressource de la fonction. Vous pouvez accorder l'autorisation d'appel à toute une API ou accorder un accès limité à une étape, à une ressource ou à une méthode.

Lorsque vous ajoutez une API à votre fonction à l'aide de la console Lambda, de la console API Gateway ou d'un modèle AWS SAM , la politique basée sur une ressource de la fonction est mise à jour automatiquement. Voici un exemple de politique de fonction.

Example stratégie de fonction

JSON

```
{
  "Version":"2012-10-17",
  "Id": "default",
  "Statement": [
    {
      "Sid": "nodejs-apig-functiongetEndpointPermissionProd-BWDBXMPLXE2F",
      "Effect": "Allow",
      "Principal": {
        "Service": "apigateway.amazonaws.com"
      },
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:us-east-2:111122223333:function:nodejs-apig-
function-1G3MXMPLXVXYI",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "111122223333"
        },
        "ArnLike": {
          "aws:SourceArn": "arn:aws:execute-api:us-
east-2:111122223333:ktyvxmpls1/*/GET/"
        }
      }
    }
  ]
}
```

Vous pouvez gérer manuellement les autorisations de politique de fonction à l'aide des opérations d'API suivantes :

- AddPermission
- RemovePermission
- GetPolicy

Pour accorder l'autorisation d'invocation à une API existante, utilisez la commande `add-permission`. Exemple :

```
aws lambda add-permission \
  --function-name my-function \
  --statement-id apigateway-get --action lambda:InvokeFunction \
  --principal apigateway.amazonaws.com \
  --source-arn "arn:aws:execute-api:us-east-2:123456789012:mnh1xmpli7/default/GET/"
```

Vous devriez voir la sortie suivante :

```
{
    "Statement": "{\"Sid\":\"apigateway-test-2\",\"Effect\":\"Allow\",\"Principal
\":{\"Service\":\"apigateway.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction
\",\"Resource\":\"arn:aws:lambda:us-east-2:123456789012:function:my-function
\",\"Condition\":{\"ArnLike\":{\"AWS:SourceArn\":\"arn:aws:execute-api:us-
east-2:123456789012:mnh1xmpli7/default/GET\"}}}"
}
```

> **Note**
>
> Si votre fonction et votre API sont différentes Régions AWS, l'identifiant de région dans l'ARN source doit correspondre à la région de la fonction, et non à la région de l'API. Quand API Gateway appelle une fonction, elle utilise un ARN de ressource basé sur l'ARN de l'API, mais modifié pour correspondre à la Région de la fonction.

L'ARN source dans cet exemple accorde l'autorisation à une intégration sur la méthode GET de la ressource racine au cours de l'étape par défaut d'une API, avec l'ID `mnh1xmpli7`. Vous pouvez utiliser un astérisque dans l'ARN source pour accorder des autorisations à plusieurs étapes, méthodes ou ressources.

Modèles de ressources

- `mnh1xmpli7/*/GET/*` – Méthode GET sur toutes les ressources à toutes les étapes.
- `mnh1xmpli7/prod/ANY/user` – Méthode ANY sur la ressource `user` à l'étape `prod`.
- `mnh1xmpli7/*/*/*` – Toute méthode sur toutes les ressources à toutes les étapes.

Pour de plus amples informations sur l'affichage de la politique et la suppression des instructions, veuillez consulter Voir les politiques IAM basées sur les ressources dans Lambda.

Exemple d'application

L'exemple d'application API Gateway with Node.js inclut une fonction avec un AWS SAM modèle qui crée une API REST dont le AWS X-Ray suivi est activé. Il inclut également des scripts pour le déploiement, l'invocation de la fonction, le test de l'API et le nettoyage.

Le gestionnaire d'événements de Powertools pour Lambda AWS

Le gestionnaire d'événements du kit d'outils Powertools for AWS Lambda fournit le routage, le middleware, la configuration CORS, la génération de spécifications OpenAPI, la validation des demandes, la gestion des erreurs et d'autres fonctionnalités utiles lors de l'écriture de fonctions Lambda invoquées par un point de terminaison API Gateway (HTTP ou REST). L'utilitaire de gestion d'événements est disponible pour Python et TypeScript/JavaScript. Pour plus d'informations, consultez l'API REST du gestionnaire d'événements dans la documentation de Powertools pour AWS Lambda (Python) et l'API REST du gestionnaire d'événements dans la documentation de Powertools pour AWS Lambda (). TypeScript

Python

```
from aws_lambda_powertools import Logger
from aws_lambda_powertools.event_handler import APIGatewayRestResolver
from aws_lambda_powertools.logging import correlation_paths
from aws_lambda_powertools.utilities.typing.lambda_context import LambdaContext

app = APIGatewayRestResolver()
logger = Logger()

@app.get("/healthz")
def ping():
    return {"message": "health status ok"}

@logger.inject_lambda_context(correlation_id_path=correlation_paths.API_GATEWAY_REST)
def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)
```

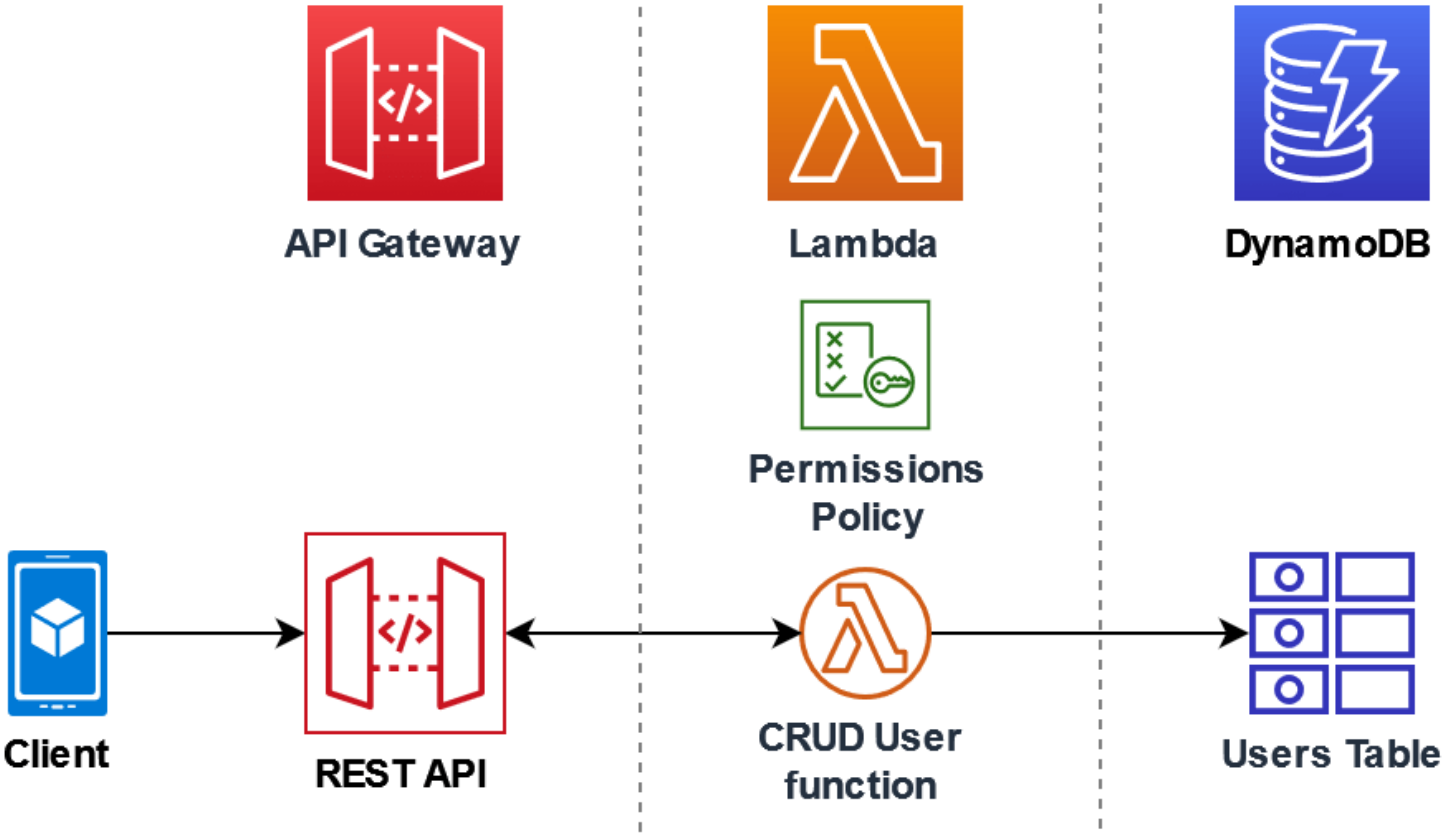
TypeScript

```
import { Router } from '@aws-lambda-powertools/event-handler/experimental-rest';
```

```
import { Logger } from '@aws-lambda-powertools/logger';
import {
  correlationPaths,
  search,
} from '@aws-lambda-powertools/logger/correlationId';
import type { Context } from 'aws-lambda/handler';

const logger = new Logger({
  correlationIdSearchFn: search,
});

const app = new Router({ logger });

app.get("/healthz", async () => {
  return { message: "health status ok" };
});

export const handler = async (event: unknown, context: Context) => {
  // You can continue using other utilities just as before
  logger.addContext(context);
  logger.setCorrelationId(event, correlationPaths.API_GATEWAY_REST);
  return app.resolve(event, context);
};
```

Didacticiel : Utiliser Lambda avec Amazon API Gateway

Dans ce tutoriel, vous créez une API REST par laquelle vous invoquez une fonction Lambda à l'aide d'une demande HTTP. Votre fonction Lambda effectuera des opérations de création, lecture, mise à jour et suppression (CRUD) sur une table DynamoDB. Cette fonction est fournie ici à titre de démonstration, mais vous apprendrez à configurer une API REST API Gateway qui peut invoquer n'importe quelle fonction Lambda.



L'utilisation d'API Gateway fournit aux utilisateurs un point de terminaison HTTP sécurisé pour invoquer votre fonction Lambda et peut aider à gérer de gros volumes d'appels à votre fonction en limitant le trafic et en validant et autorisant automatiquement les appels d'API. API Gateway fournit également des contrôles de sécurité flexibles à l'aide de Gestion des identités et des accès AWS (IAM) et d'Amazon Cognito. Ceci est utile pour les cas d'utilisation où une autorisation préalable est requise pour les appels à votre application.

> **Tip**
>
> Lambda propose deux méthodes pour appeler votre fonction via un point de terminaison HTTP : API Gateway et fonction Lambda. URLs Si vous ne savez pas quelle est la meilleure méthode pour votre cas d'utilisation, consultez the section called "API Gateway et fonction URLs".

Pour réaliser ce tutoriel, vous passerez par les étapes suivantes :

1. Créer et configurer une fonction Lambda en Python ou Node.js pour effectuer des opérations sur une table DynamoDB.

2. Créer une API REST dans API Gateway pour se connecter à votre fonction Lambda.
3. Créer une table DynamoDB et la tester avec votre fonction Lambda dans la console.
4. Déployer votre API et tester la configuration complète en utilisant curl dans un terminal.

En suivant ces étapes, vous apprendrez à utiliser API Gateway pour créer un point de terminaison HTTP capable d'invoquer en toute sécurité une fonction Lambda à n'importe quelle échelle. Vous apprendrez également à déployer votre API, et à la tester dans la console et en envoyant une demande HTTP à l'aide d'un terminal.

Création d'une stratégie d'autorisations

Avant de créer un rôle d'exécution pour votre fonction Lambda, vous devez d'abord créer une politique d'autorisation pour autoriser votre fonction à accéder aux ressources requises AWS . Pour ce didacticiel, la politique permet à Lambda d'effectuer des opérations CRUD sur une table DynamoDB et d'écrire sur Amazon Logs. CloudWatch

Pour créer la politique

1. Ouvrez la page stratégies de la console IAM.
2. Choisissez Créer une stratégie.
3. Choisissez l'onglet JSON, puis collez la stratégie personnalisée suivante dans l'éditeur JSON.

   JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt1428341300017",
      "Action": [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem"
      ],
      "Effect": "Allow",
      "Resource": "*"
```

```
    },
    {
      "Sid": "",
      "Resource": "*",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Effect": "Allow"
    }
  ]
}
```

4. Choisissez Suivant : Balises.
5. Choisissez Suivant : Vérification.
6. Sous Examiner une stratégie, pour le Nom de la stratégie, saisissez **`lambda-apigateway-policy`**.
7. Choisissez Créer une stratégie.

Créer un rôle d’exécution

Un rôle d'exécution est un rôle Gestion des identités et des accès AWS (IAM) qui accorde à une fonction Lambda l'autorisation d' Services AWS accès et de ressources. Pour permettre à votre fonction d’effectuer des opérations sur une table DynamoDB, vous attachez la politique d’autorisation que vous avez créée à l’étape précédente.

Pour créer un rôle d’exécution et attacher votre politique d’autorisations personnalisée

1. Ouvrez la page Rôles de la console IAM.
2. Sélectionnez Créer un rôle.
3. Pour le type d’entité de confiance, choisissez Service AWS , puis pour le cas d’utilisation, choisissez Lambda.
4. Choisissez Suivant.
5. Dans la zone de recherche de stratégie, entrez **`lambda-apigateway-policy`**.
6. Dans les résultats de la recherche, sélectionnez la stratégie que vous avez créée (`lambda-apigateway-policy`), puis choisissez Suivant.

7. Sous Role details (Détails du rôle), pour Role name (Nom du rôle), saisissez **`lambda-apigateway-role`**, puis sélectionnez Create role (Créer un rôle).

Créer la fonction Lambda

1. Ouvrez la page Fonctions de la console Lambda et choisissez Créer une fonction.
2. Choisissez Créer à partir de zéro.
3. Sous Nom de la fonction, saisissez `LambdaFunctionOverHttps`.
4. Pour Exécution, choisissez le dernier environnement d'exécution Node.js ou Python.
5. Sous Permissions (Autorisations), développez Change default execution role (Modifier le rôle d'exécution par défaut).
6. Sélectionnez Utiliser un rôle existant, sélectionnez le rôle **`lambda-apigateway-role`** que vous avez créé précédemment.
7. Choisissez Créer une fonction.
8. Dans le volet Source du code, remplacez le code par défaut par le code Node.js ou Python suivant.

   Node.js

   Le `region` paramètre doit correspondre à l' Région AWS endroit où vous déployez la fonction et créez la table DynamoDB.

   Example index.mjs

```
import { DynamoDBDocumentClient, PutCommand, GetCommand,
         UpdateCommand, DeleteCommand} from "@aws-sdk/lib-dynamodb";
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

const ddbClient = new DynamoDBClient({ region: "us-east-2" });
const ddbDocClient = DynamoDBDocumentClient.from(ddbClient);

// Define the name of the DDB table to perform the CRUD operations on
const tablename = "lambda-apigateway";

/**
 * Provide an event that contains the following keys:
 *
 *   - operation: one of 'create,' 'read,' 'update,' 'delete,' or 'echo'
```

```
 *   - payload: a JSON object containing the parameters for the table item
 *     to perform the operation on
 */
export const handler = async (event, context) => {

    const operation = event.operation;

    if (operation == 'echo'){
         return(event.payload);
    }

   else {
       event.payload.TableName = tablename;
       let response;

       switch (operation) {
         case 'create':
              response = await ddbDocClient.send(new
 PutCommand(event.payload));
              break;
         case 'read':
              response = await ddbDocClient.send(new
 GetCommand(event.payload));
              break;
         case 'update':
              response = ddbDocClient.send(new UpdateCommand(event.payload));
              break;
         case 'delete':
              response = ddbDocClient.send(new DeleteCommand(event.payload));
              break;
         default:
           response = 'Unknown operation: ${operation}';
         }
       console.log(response);
       return response;
   }
};
```

Python

Example lambda_function.py

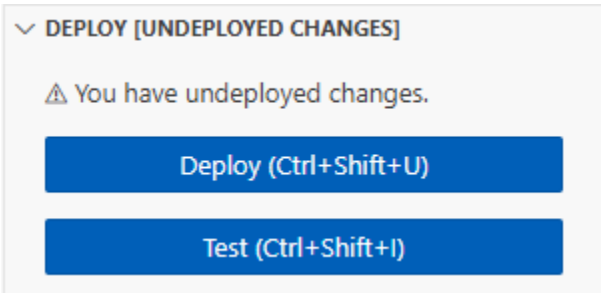
```
import boto3
```

```
# Define the DynamoDB table that Lambda will connect to
table_name = "lambda-apigateway"

# Create the DynamoDB resource
dynamo = boto3.resource('dynamodb').Table(table_name)

# Define some functions to perform the CRUD operations
def create(payload):
    return dynamo.put_item(Item=payload['Item'])

def read(payload):
    return dynamo.get_item(Key=payload['Key'])

def update(payload):
    return dynamo.update_item(**{k: payload[k] for k in ['Key',
 'UpdateExpression',
    'ExpressionAttributeNames', 'ExpressionAttributeValues'] if k in payload})

def delete(payload):
    return dynamo.delete_item(Key=payload['Key'])

def echo(payload):
    return payload

operations = {
    'create': create,
    'read': read,
    'update': update,
    'delete': delete,
    'echo': echo,
}

def lambda_handler(event, context):
    '''Provide an event that contains the following keys:
      - operation: one of the operations in the operations dict below
      - payload: a JSON object containing parameters to pass to the
        operation being performed
    '''

    operation = event['operation']
    payload = event['payload']

    if operation in operations:
```

```
        return operations[operation](payload)

    else:
        raise ValueError(f'Unrecognized operation "{operation}"')
```

> **Note**
>
> Dans cet exemple, le nom de la table DynamoDB est défini comme une variable dans le code de votre fonction. Dans une application réelle, passer ce paramètre comme une variable d'environnement et éviter de coder en dur le nom de la table constitue une bonne pratique. Pour plus d'informations, consultez la section Utilisation de variables d' AWS Lambda environnement.

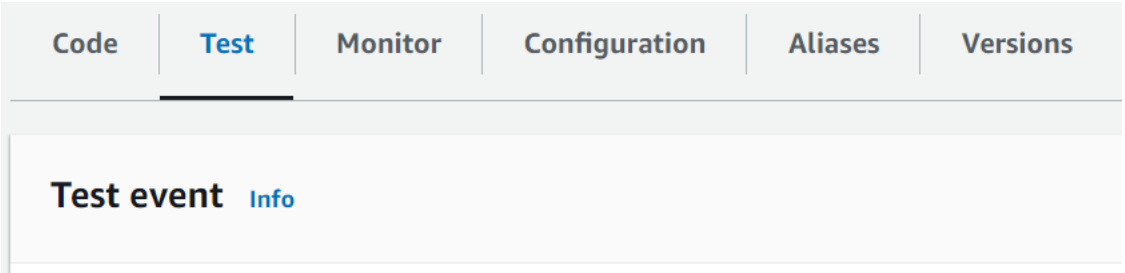
9. Dans la section DÉPLOYER, choisissez Déployer pour mettre à jour le code de votre fonction :



Tester la fonction

Avant d'intégrer votre fonction à API Gateway, confirmez que vous avez déployé la fonction avec succès. Utilisez la console Lambda pour envoyer un événement de test à votre fonction.

1. Dans la page de votre fonction de la console Lambda, choisissez l'onglet Tester.



2. Défilez jusqu'à la section JSON d'événement et remplacez l'événement par défaut par ce qui suit. Cet événement correspond à la structure attendue par la fonction Lambda.

```
{
    "operation": "echo",
    "payload": {
        "somekey1": "somevalue1",
        "somekey2": "somevalue2"
    }
}
```

3. Sélectionnez Tester).
4. Sous Exécution de la fonction : succès, développez Détails. Vous devriez voir la réponse suivante :

```
{
  "somekey1": "somevalue1",
  "somekey2": "somevalue2"
}
```

Créer une API REST avec API Gateway

Dans cette étape, vous créez l'API REST API Gateway que vous utiliserez pour invoquer votre fonction Lambda.

Pour créer l'API

1. Ouvrez la console API Gateway.
2. Sélectionnez Create API (Créer une API).
3. Dans la boîte de dialogue API REST, choisissez Créer.
4. Sous Détails de l'API, laissez Nouvelle API sélectionnée et pour Nom de l'API, saisissez **`DynamoDBOperations`**.
5. Sélectionnez Create API (Créer une API).

Créez une ressource sur votre API REST

Pour ajouter une méthode HTTP à votre API, vous devez d'abord créer une ressource pour que cette méthode puisse fonctionner. Ici, vous créez la ressource pour gérer votre table DynamoDB.

Pour créer la ressource

1. Dans la console API Gateway, sur la page Ressources de votre API, choisissez Créer une ressource.
2. Dans Détails de la ressource, pour Nom de la ressource saisissez **`DynamoDBManager`**.
3. Choisissez Create Resource.

Création d'une méthode HTTP POST

Dans cette étape, vous créez une méthode (`POST`) pour votre ressource `DynamoDBManager`. Vous liez cette méthode `POST` à votre fonction Lambda de sorte que lorsque la méthode reçoit une demande HTTP, API Gateway invoque votre fonction Lambda.

> **Note**
>
> Dans le cadre de ce tutoriel, une méthode HTTP (`POST`) est utilisée pour invoquer une seule fonction Lambda qui exécute toutes les opérations sur votre table DynamoDB. Dans une application réelle, l'utilisation d'une fonction Lambda et d'une méthode HTTP différentes pour chaque opération constitue une bonne pratique. Pour plus d'informations, consultez Le monolithe Lambda dans Serverless Land.

Pour créer la méthode POST

1. Sur la page Ressources de votre API, assurez-vous que la ressource `/DynamoDBManager` est surlignée. Ensuite, dans le volet Méthodes, choisissez Créer une méthode.
2. Pour Type de méthode, sélectionnez POST.
3. Pour Type d'intégration, laissez la Fonction Lambda sélectionnée.
4. Pour la Fonction Lambda, choisissez l'Amazon Resource Name (ARN) pour votre fonction (`LambdaFunctionOverHttps`).
5. Choisissez Créer une méthode.

Créez une table DynamoDB

Créez une table DynamoDB vide sur laquelle votre fonction Lambda effectuera des opérations CRUD.

Créer le tableau DynamoDB

1. Ouvrez la page Tables (Tables) de la console DynamoDB.
2. Choisissez Créer un tableau.
3. Sous Détails du tableau, procédez comme suit :
    1. Sous Nom du tableau, saisissez **`lambda-apigateway`**.
    2. Pour Clé de partition, saisissez **`id`**, et conservez le type de données défini comme Chaîne.
4. Sous Table settings (Paramètres de la table), conservez les Default settings (Paramètres par défaut).
5. Choisissez Créer un tableau.

Test de l'intégration d'API Gateway, Lambda et DynamoDB

Vous êtes maintenant prêt à tester l'intégration de votre méthode API Gateway avec votre fonction Lambda et votre table DynamoDB. À l'aide de la console API Gateway, vous envoyez des demandes directement à votre méthode `POST` en utilisant la fonction de test de la console. Dans cette étape, vous utilisez d'abord une opération `create` pour ajouter un nouvel élément à votre table DynamoDB, puis vous utilisez une opération `update` pour modifier l'élément.

Test 1 : Pour créer un nouvel élément dans votre table DynamoDB

1. Dans la console API Gateway, choisissez votre API (`DynamoDBOperations`).
2. Choisissez la méthode POST sous la ressource `DynamoDBManager`.
3. Choisissez l'onglet Test. Vous devrez peut-être choisir la flèche droite pour afficher l'onglet.
4. Sous Méthode de test, laissez les Chaînes de requête et les En-têtes vides. Pour Corps de requête, collez l'élément JSON suivant :

```
{
  "operation": "create",
  "payload": {
    "Item": {
      "id": "1234ABCD",
      "number": 5
    }
  }
}
```

5. Sélectionnez Tester).

   Les résultats qui s'affichent à la fin du test doivent indiquer le statut `200`. Ce code d'état indique que l'opération `create` a réussi.

   Pour confirmer, vérifiez que votre table DynamoDB contient maintenant le nouvel élément.

6. Ouvrez la page Tables de la console DynamoDB et choisissez la table `lambda-apigateway`.
7. Choisissez Explore table items (Explorer les éléments de la table). Dans le volet Items returned (Éléments retournés), vous devriez voir un élément avec l'id `1234ABCD` et le numéro `5`. Exemple :

Test 2 : Pour mettre à jour l'élément dans votre table DynamoDB

1. Dans la Console API Gateway, revenez à l'onglet Test votre méthode POST.
2. Sous Méthode de test, laissez les Chaînes de requête et les En-têtes vides. Pour Corps de requête, collez l'élément JSON suivant :

```
{
    "operation": "update",
    "payload": {
        "Key": {
            "id": "1234ABCD"
        },
        "UpdateExpression": "SET #num = :newNum",
        "ExpressionAttributeNames": {
            "#num": "number"
        },
        "ExpressionAttributeValues": {
            ":newNum": 10
        }
    }
```

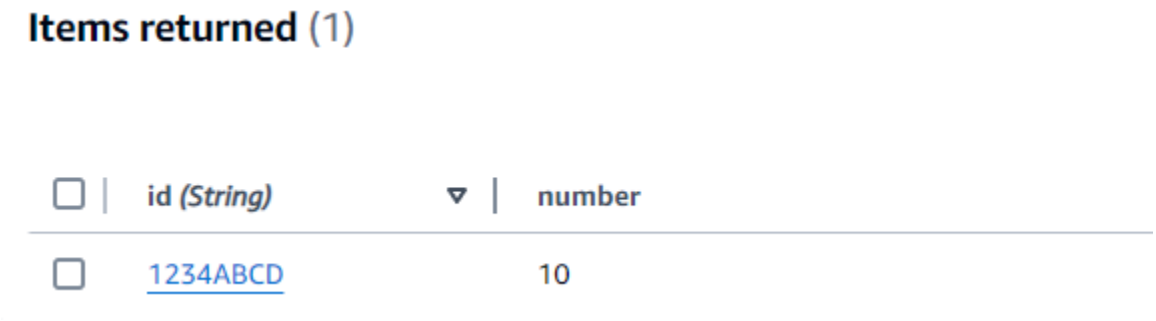
```
}
```

3. Sélectionnez Tester).

   Les résultats qui s'affichent à la fin du test devraient montrer l'état `200`. Ce code d'état indique que l'opération `update` a réussi.

   Pour confirmer, vérifiez que l'élément dans votre table DynamoDB a été modifié.

4. Ouvrez la page Tables de la console DynamoDB et choisissez la table `lambda-apigateway`.
5. Choisissez Explore table items (Explorer les éléments de la table). Dans le volet Items returned (Éléments retournés), vous devriez voir un élément avec l'id `1234ABCD` et le numéro `10`.



Déploiement de l'API

Pour qu'un client puisse appeler l'API, vous devez créer un déploiement et une étape associée. Une étape représente un instantané de votre API, y compris ses méthodes et intégrations.

Pour déployer l'API

1. Ouvrez la APIspage de la console API Gateway et choisissez l'`DynamoDBOperations`API.
2. Sur la page Ressources de votre API, choisissez Deploy API (Déployer l'API).
3. Pour Stage (Étape), choisissez *New stage* (Nouvelle étape), puis pour Stage name (Nom de l'étape), saisissez **`test`**.
4. Choisissez Déployer.
5. Dans le volet Stage details (Détails de l'étape), copiez Invoke URL (URL d'invocation). Vous l'utiliserez à l'étape suivante pour invoquer votre fonction à l'aide d'une demande HTTP.

Utilisez curl pour invoquer votre fonction à l'aide de demandes HTTP

Vous pouvez maintenant invoquer votre fonction Lambda en émettant une demande HTTP vers votre API. Dans cette étape, vous allez créer un nouvel élément dans votre table DynamoDB, puis effectuer des oéprations de lecture, de mise à jour et de suppression sur cet élément.

Pour créer un élément dans votre table DynamoDB à l'aide de curl

1. Ouvrez un terminal ou une invite de commande sur votre machine locale et exécutez la commande `curl` suivante en utilisant l'URL d'invocation que vous avez copiée à l'étape précédente. Cette commande utilise les options suivantes :

   - `-H` : ajoute un en-tête personnalisé à la demande. Ici, le paramètre spécifie le type de contenu comme étant du JSON.
   - `-d` : envoie des données dans le corps de la demande. Cette option utilise une méthode HTTP POST par défaut.

   Linux/macOS

   ```
   curl https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager
    \
   -H "Content-Type: application/json" \
   -d '{"operation": "create", "payload": {"Item": {"id": "5678EFGH", "number":
    15}}}'
   ```

   PowerShell

   ```
   curl.exe 'https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/
   DynamoDBManager' -H 'Content-Type: application/json' -d '{\"operation\":
    \"create\", \"payload\": {\"Item\": {\"id\": \"5678EFGH\", \"number\": 15}}}'
   ```

   Si l'opération a réussi, vous devriez voir une réponse renvoyée avec un code d'état HTTP de 200.

2. Vous pouvez également utiliser la console DynamoDB pour vérifier que le nouvel élément figure dans votre table en procédant comme suit :

   1. Ouvrez la page Tables de la console DynamoDB et choisissez la table `lambda-apigateway`.

2. Sélectionnez Explore table items (Explorer les éléments de la table). Dans le volet Items returned (Éléments retournés), vous devriez voir un élément avec l'id `5678EFGH` et le numéro `15`.

Pour lire l'élément dans votre table DynamoDB à l'aide de curl

- Dans votre terminal ou invite de commande, exécutez la commande `curl` suivante pour lire la valeur l'élément que vous venez de créer. Utilisez votre propre URL invoquée.

  Linux/macOS

  ```
  curl https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager
   \
  -H "Content-Type: application/json" \
  -d '{"operation": "read", "payload": {"Key": {"id": "5678EFGH"}}}'
  ```

  PowerShell

  ```
  curl.exe 'https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/
  DynamoDBManager' -H 'Content-Type: application/json' -d '{\"operation\": \"read
  \", \"payload\": {\"Key\": {\"id\": \"5678EFGH\"}}}'
  ```

  Vous devriez obtenir un résultat semblable à l'un des suivants selon que vous avez choisi le code de fonction Node.js ou Python :

  Node.js

  ```
  {"$metadata":
  {"httpStatusCode":200,"requestId":"7BP3G5Q0C0O1E50FBQI9NS099JVV4KQNSO5AEMVJF66Q9ASUAAJG"
  "attempts":1,"totalRetryDelay":0},"Item":{"id":"5678EFGH","number":15}}
  ```

  Python

  ```
  {"Item":{"id":"5678EFGH","number":15},"ResponseMetadata":
  {"RequestId":"QNDJICE52E86B82VETR6RKBE5BVV4KQNSO5AEMVJF66Q9ASUAAJG",
  "HTTPStatusCode":200,"HTTPHeaders":{"server":"Server","date":"Wed, 31 Jul 2024
   00:37:01 GMT","content-type":"application/x-amz-json-1.0",
  ```

```
"content-length":"52","connection":"keep-alive","x-amzn-
requestid":"QNDJICE52E86B82VETR6RKBE5BVV4KQNSO5AEMVJF66Q9ASUAAJG","x-amz-
crc32":"2589610852"},
"RetryAttempts":0}}
```

Pour mettre à jour l'élément dans votre table DynamoDB à l'aide de curl

1. Dans votre terminal ou invite de commande, exécutez la commande `curl` suivante pour mettre à jour l'élément que vous venez de créer en modifiant la valeur `number`. Utilisez votre propre URL invoquée.

   Linux/macOS

   ```
   curl https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager
    \
   -H "Content-Type: application/json" \
   -d '{"operation": "update", "payload": {"Key": {"id": "5678EFGH"},
    "UpdateExpression": "SET #num = :new_value", "ExpressionAttributeNames":
    {"#num": "number"}, "ExpressionAttributeValues": {":new_value": 42}}}'
   ```

   PowerShell

   ```
   curl.exe 'https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/
   DynamoDBManager' -H 'Content-Type: application/json' -d '{\"operation\":
    \"update\", \"payload\": {\"Key\": {\"id\": \"5678EFGH\"}, \"UpdateExpression
   \": \"SET #num = :new_value\", \"ExpressionAttributeNames\": {\"#num\": \"number
   \"}, \"ExpressionAttributeValues\": {\":new_value\": 42}}}'
   ```

2. Pour confirmer que la valeur de `number` de l'élément a été mise à jour, exécutez une autre commande de lecture :

   Linux/macOS

   ```
   curl https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager
    \
   -H "Content-Type: application/json" \
   -d '{"operation": "read", "payload": {"Key": {"id": "5678EFGH"}}}'
   ```

PowerShell

```
curl.exe 'https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/
DynamoDBManager' -H 'Content-Type: application/json' -d '{\"operation\": \"read
\", \"payload\": {\"Key\": {\"id\": \"5678EFGH\"}}}'
```

Pour supprimer l'élément dans votre table DynamoDB à l'aide de curl

1. Dans votre terminal ou invite de commande, exécutez la commande `curl` suivante pour supprimer l'élément que vous venez de créer. Utilisez votre propre URL invoquée.

   Linux/macOS

   ```
   curl https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager
    \
   -H "Content-Type: application/json" \
   -d '{"operation": "delete", "payload": {"Key": {"id": "5678EFGH"}}}'
   ```

   PowerShell

   ```
   curl.exe 'https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/
   DynamoDBManager' -H 'Content-Type: application/json' -d '{\"operation\":
    \"delete\", \"payload\": {\"Key\": {\"id\": \"5678EFGH\"}}}'
   ```

2. Confirmez que l'opération de suppression a réussi. Dans le volet Items returned (Éléments retournés) de la page Explore items (Explorer les éléments) de la console DynamoDB, vérifiez que l'élément avec l'id `5678EFGH` n'est plus dans la table.

Nettoyer vos ressources (facultatif)

Vous pouvez maintenant supprimer les ressources que vous avez créées pour ce didacticiel, sauf si vous souhaitez les conserver. En supprimant AWS les ressources que vous n'utilisez plus, vous évitez des frais inutiles pour votre Compte AWS.

Pour supprimer la fonction Lambda

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction que vous avez créée.

3. Sélectionnez Actions, Supprimer.
4. Saisissez **`confirm`** dans la zone de saisie de texte et choisissez Delete (Supprimer).

Pour supprimer le rôle d'exécution

1. Ouvrez la page Roles (Rôles) de la console IAM.
2. Sélectionnez le rôle d'exécution que vous avez créé.
3. Sélectionnez Delete (Supprimer).
4. Saisissez le nom du rôle dans le champ de saisie de texte et choisissez Delete (Supprimer).

Pour supprimer l'API

1. Ouvrez la APIs page de la console API Gateway.
2. Sélectionnez l'API que vous avez créée.
3. Sélectionnez Actions, Supprimer.
4. Sélectionnez Supprimer.

Pour supprimer la table DynamoDB

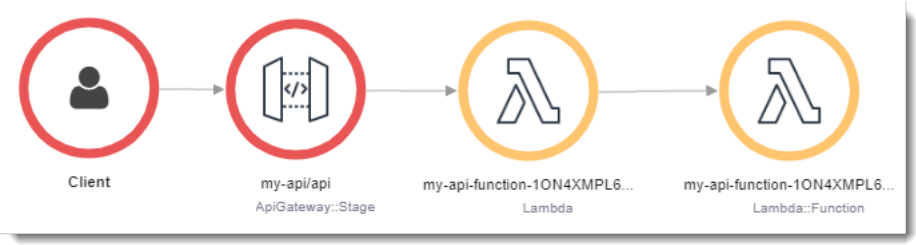
1. Ouvrez la page Tables (Tables) de la console DynamoDB.
2. Sélectionnez la table que vous avez créée.
3. Choisissez Supprimer.
4. Saisissez **`delete`** dans la zone de texte.
5. Choisissez Supprimer la table.

Gestion des erreurs Lambda avec une API Gateway

API Gateway traite toutes les erreurs d'invocation et de fonction comme des erreurs internes. Si l'API Lambda rejette la demande d'invocation, API Gateway renvoie un code d'erreur 500. Si la fonction s'exécute mais renvoie une erreur ou une réponse dans un format inadéquat, API Gateway renvoie un code d'erreur 502. Dans les deux cas, le corps de la réponse d'API Gateway est `{"message": "Internal server error"}`.

> **Note**
>
> API Gateway ne fait aucune nouvelle tentative d'appel Lambda. Si Lambda renvoie une erreur, API Gateway renvoie une réponse d'erreur au client.

L'exemple suivant montre une cartographie de suivi X-Ray pour une demande qui a entraîné une erreur de fonction et un message d'erreur 502 d'API Gateway. Le client reçoit le message d'erreur générique.

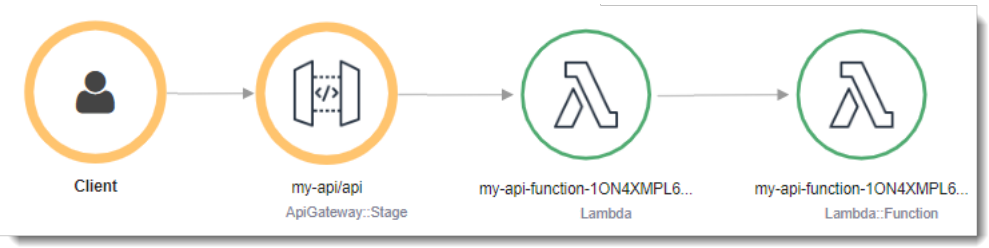


Pour personnaliser la réponse d'erreur, vous devez attraper les erreurs dans votre code et formater une réponse dans le format requis.

Example index.js : formatage des erreurs

```
var formatError = function(error){
  var response = {
    "statusCode": error.statusCode,
    "headers": {
      "Content-Type": "text/plain",
      "x-amzn-ErrorType": error.code
    },
    "isBase64Encoded": false,
    "body": error.code + ": " + error.message
  }
  return response
}
```

API Gateway convertit cette réponse en une erreur HTTP avec un code d'état et un corps personnalisés. Dans la carte de trace, le nœud de fonction est vert car il a géré l'erreur.



Sélection d'une méthode pour invoquer votre fonction Lambda à l'aide d'une requête HTTP

De nombreux cas d'utilisation courants de Lambda impliquent d'invoquer votre fonction à l'aide d'une requête HTTP. Vous pourriez souhaiter qu'une application Web invoque votre fonction par le biais d'une demande de navigateur. Les fonctions Lambda peuvent également être utilisées pour créer un REST complet APIs, gérer les interactions des utilisateurs à partir d'applications mobiles, traiter des données provenant de services externes via des appels HTTP ou créer des webhooks personnalisés.

Les sections suivantes expliquent quels sont vos choix pour invoquer Lambda via HTTP et fournissent des informations qui vous aideront à prendre la bonne décision pour votre cas d'utilisation particulier.

Options proposées lors de la sélection d'une méthode d'appel HTTP

Lambda propose deux méthodes principales pour appeler une fonction à l'aide d'une requête HTTP : function URLs et API Gateway. Les principales différences entre ces deux options sont les suivantes :

- La fonction Lambda URLs fournit un point de terminaison HTTP simple et direct pour une fonction Lambda. Elles sont optimisées pour la simplicité et la rentabilité et constituent le moyen le plus rapide d'exposer une fonction Lambda via HTTP.
- API Gateway est un service plus avancé permettant de créer des fonctionnalités complètes APIs. API Gateway est optimisé pour créer et gérer des productions APIs à grande échelle et fournit des outils complets pour la sécurité, la surveillance et la gestion du trafic.

Recommandations si vous connaissez déjà vos besoins

Si vous connaissez déjà bien vos besoins, voici nos recommandations de base :

Nous recommandons cette fonction URLs pour les applications simples ou le prototypage où vous n'avez besoin que de méthodes d'authentification et de request/response manipulation de base et où vous souhaitez réduire les coûts et la complexité au minimum.

API Gateway est un meilleur choix pour les applications de production à grande échelle ou pour les cas où vous avez besoin de fonctionnalités plus avancées telles que la prise en charge d'OpenAPI Description, un choix d'options d'authentification, des noms de domaine personnalisés ou une request/response gestion riche, notamment la régulation, la mise en cache et la transformation. request/response

Éléments à prendre en compte lors de la sélection d'une méthode pour invoquer votre fonction Lambda

Lorsque vous choisissez entre function URLs et API Gateway, vous devez prendre en compte les facteurs suivants :

- Vos besoins en matière d'authentification, par exemple si vous avez besoin OAuth d'Amazon Cognito pour authentifier les utilisateurs
- Vos exigences en matière de mise à l'échelle et la complexité de l'API que vous souhaitez implémenter
- Si vous avez besoin de fonctionnalités avancées telles que la validation et le request/response formatage des demandes
- Vos besoins en matière de surveillance
- Vos objectifs en matière de coûts

La compréhension de ces facteurs vous permettra de choisir l'option qui répond le mieux à vos exigences en matière de sécurité, de complexité et de coût.

Le tableau suivant résume les différences entre les deux options.

Authentification

- La fonction URLs fournit des options d'authentification de base via Gestion des identités et des accès AWS (IAM). Vous pouvez configurer vos points de terminaison pour qu'ils soient publics (aucune authentification) ou pour qu'ils nécessitent une authentification IAM. Avec l'authentification IAM, vous pouvez utiliser des AWS informations d'identification standard ou des rôles IAM pour contrôler l'accès. Bien que simple à configurer, cette approche offre des options limitées par rapport aux autres méthodes d'authentification.
- API Gateway donne accès à une gamme plus complète d'options d'authentification. Outre l'authentification IAM, vous pouvez utiliser des autorisateurs Lambda (logique d'authentification personnalisée), des groupes d'utilisateurs Amazon Cognito et des flux .0. OAuth2 Cette flexibilité

vous permet de mettre en œuvre des schémas d'authentification complexes, notamment des fournisseurs d'authentification tiers, une authentification basée sur des jetons et une authentification multifactorielle.

Traitement des requêtes et réponses

- La fonction URLs fournit une gestion de base des requêtes et des réponses HTTP. Elles prennent en charge les méthodes HTTP standard et créent une prise en charge intégrée du partage des ressources entre origines (CORS, cross-origin resource sharing). Bien qu'elles puissent gérer naturellement les charges utiles JSON et les paramètres de requête, elles n'offrent pas de fonctionnalités de transformation ou de validation des requêtes. La gestion des réponses est tout aussi simple : le client reçoit la réponse de votre fonction Lambda exactement telle que Lambda la renvoie.
- API Gateway fournit des fonctionnalités sophistiquées de gestion des requêtes et des réponses. Vous pouvez définir des validateurs de demandes, transformer les demandes et les réponses à l'aide de modèles de mappage, configurer request/response des en-têtes et implémenter la mise en cache des réponses. API Gateway prend également en charge les charges utiles binaires et les noms de domaine personnalisés et peut modifier les réponses avant qu'elles n'atteignent le client. Vous pouvez configurer des modèles pour la request/response validation et la transformation à l'aide du schéma JSON.

Mise à l'échelle

- Les fonctions s' URLsadaptent directement aux limites de simultanéité de votre fonction Lambda et gérez les pics de trafic en augmentant la taille de votre fonction jusqu'à sa limite de simultanéité maximale configurée. Une fois cette limite atteinte, Lambda répond aux requêtes supplémentaires avec des réponses HTTP 429. Il n'existe aucun mécanisme de mise en file d'attente intégré. La gestion de la mise à l'échelle dépend donc entièrement de la configuration de votre fonction Lambda. Par défaut, les fonctions Lambda sont limitées à 1 000 exécutions simultanées par fonction. Région AWS
- API Gateway fournit des fonctionnalités de dimensionnement supplémentaires en plus de la propre mise à l'échelle de Lambda. Il inclut des commandes intégrées de mise en file d'attente et de limitation des requêtes, ce qui vous permet de gérer les pics de trafic de manière plus élégante. API Gateway peut traiter jusqu'à 10 000 requêtes par seconde et par région par défaut, avec une capacité de débordement de 5 000 requêtes par seconde. Il fournit également des outils pour limiter les requêtes à différents niveaux (API, étape ou méthode) afin de protéger votre dorsal.

Contrôle

- Les fonctions URLs offrent une surveillance de base via CloudWatch les métriques Amazon, notamment le nombre de demandes, la latence et les taux d'erreur. Vous avez accès aux métriques et aux journaux Lambda standard, qui indiquent les requêtes brutes entrant dans votre fonction. Bien que cela fournisse une visibilité opérationnelle essentielle, les métriques se concentrent principalement sur l'exécution des fonctions.
- API Gateway fournit des fonctionnalités de surveillance complètes, notamment des métriques détaillées, des options de journalisation et de suivi. Vous pouvez surveiller les appels d'API, la latence, les taux d'erreur et hit/miss les taux de cache via CloudWatch. API Gateway s'intègre également au AWS X-Ray traçage distribué et fournit des formats de journalisation personnalisables.

Cost

- Les fonctions URLs suivent le modèle de tarification Lambda standard : vous ne payez que pour les appels de fonctions et le temps de calcul. Il n'y a pas de frais supplémentaires pour le point de terminaison de l'URL lui-même. Cela en fait un choix rentable pour les applications simples APIs ou à faible trafic si vous n'avez pas besoin des fonctionnalités supplémentaires d'API Gateway.
- API Gateway propose un niveau gratuit qui inclut un million d'appels d'API reçus pour REST APIs et un million d'appels d'API reçus pour HTTP APIs. Ensuite, API Gateway facture les appels d'API, le transfert de données et la mise en cache (si activée). Reportez-vous à la page de tarification d'API Gateway pour connaître les coûts associés à votre propre cas d'utilisation.

Autres fonctions

- URLsLes fonctions sont conçues pour la simplicité et l'intégration directe de Lambda. Ils prennent en charge les points de terminaison HTTP et HTTPS, offrent un support CORS intégré et fournissent des points de terminaison à double pile (IPv4 et IPv6). Bien qu'elles ne disposent d'aucune fonctionnalité avancée, elles excellent dans les scénarios où vous avez besoin d'un moyen rapide et simple d'exposer les fonctions Lambda via HTTP.
- API Gateway inclut de nombreuses fonctionnalités supplémentaires telles que la gestion des versions des API, la gestion des étapes, les clés d'API pour les plans d'utilisation, la documentation des API via Swagger/OpenAPI, le mode WebSocket APIs privé au sein d' APIs un VPC et l'intégration WAF pour une sécurité accrue. Il prend également en charge les déploiements Canary,

les intégrations fictives à des fins de test et l'intégration avec d'autres solutions que Services AWS Lambda.

Sélection d’une méthode pour invoquer votre fonction Lambda

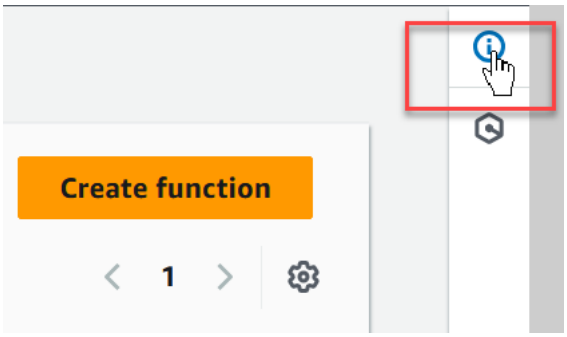
Maintenant que vous avez pris connaissance des critères de sélection entre la fonction Lambda URLs et API Gateway et des principales différences entre les deux, vous pouvez sélectionner l'option qui répond le mieux à vos besoins et utiliser les ressources suivantes pour vous aider à commencer à l'utiliser.

Function URLs

Commencez à utiliser Function URLs grâce aux ressources suivantes

- Suivre le tutoriel Création d’une fonction Lambda avec une URL de fonction
- Pour en savoir plus sur URLs les fonctions, consultez le the section called “Fonction URLs” chapitre de ce guide
- Essayez le tutoriel guidé intégré à la console Créer une application Web simple en procédant comme suit :

1. Ouvrez la page Fonctions de la console Lambda.
2. Ouvrez le panneau d’aide en cliquant sur l’icône dans le coin supérieur droit de l’écran.



3. Sélectionnez Tutoriels.
4. Dans Créer une application Web simple, choisissez Démarrer le tutoriel.

API Gateway

Mise en route avec Lambda et API Gateway grâce aux ressources suivantes

- Suivez le tutoriel Utilisation de Lambda avec API Gateway pour créer une API REST intégrée à une fonction Lambda principale.
- Pour en savoir plus sur les différents types d'API proposés par API Gateway, consultez les sections suivantes du Guide du développeur Amazon API Gateway :
  - API Gateway REST APIs
  - API Gateway HTTP APIs
  - API Gateway WebSocket APIs
- Essayez un ou plusieurs des exemples de la section Tutoriels et ateliers du manuel Guide du développeur Amazon API Gateway.

Utilisation AWS Lambda avec AWS Infrastructure Composer

AWS Infrastructure Composer est un constructeur visuel permettant de concevoir des applications modernes sur AWS. Vous concevez l'architecture de votre application en faisant glisser, en regroupant et Services AWS en connectant dans un canevas visuel. Infrastructure Composer crée des modèles d'infrastructure en tant que code (IaC) à partir de votre conception que vous pouvez déployer en utilisant AWS SAM ou CloudFormation.

Exportation d'une fonction Lambda vers Infrastructure Composer

Vous pouvez commencer à utiliser Infrastructure Composer en créant un nouveau projet basé sur la configuration d'une fonction Lambda existante à l'aide de la console Lambda. Pour exporter la configuration et le code de votre fonction vers Infrastructure Composer afin de créer un nouveau projet, procédez comme suit :

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction que doit utiliser comme base pour votre projet Infrastructure Composer.
3. Dans le volet Présentation de la fonction, choisissez Exporter vers Infrastructure Composer.

   Pour exporter la configuration et le code de votre fonction vers Infrastructure Composer, Lambda crée un compartiment Amazon S3 dans votre compte pour stocker temporairement ces données.

4. Dans la boîte de dialogue, choisissez Confirmer et créer un projet pour accepter le nom par défaut de ce compartiment et exporter la configuration et le code de votre fonction vers Infrastructure Composer.
5. (Facultatif) Pour choisir un autre nom pour le compartiment Amazon S3 créé par Lambda, entrez un nouveau nom et choisissez Confirmer et créer un projet. Les noms de compartiment Amazon S3 doivent être uniques et respecter les règles de dénomination de compartiment.
6. Pour enregistrer vos fichiers de projet et de fonction dans Infrastructure Composer, activez le mode de synchronisation locale.

> **Note**
>
> Si vous avez déjà utilisé la fonctionnalité Exporter vers Application Composer et créé un compartiment Amazon S3 en utilisant le nom par défaut, Lambda peut réutiliser ce compartiment s'il existe toujours. Acceptez le nom du compartiment par défaut dans la boîte de dialogue pour réutiliser le compartiment existant.

Configuration des compartiments de transfert Amazon S3 Transfer AccelAccelAccel

Le compartiment Amazon S3 créé par Lambda pour transférer la configuration de votre fonction chiffre automatiquement les objets à l'aide de la norme de chiffrement AES 256. Lambda configure également le bucket pour utiliser la condition de propriétaire du bucket afin de garantir que vous Compte AWS êtes le seul à pouvoir ajouter des objets au bucket.

Lambda configure le compartiment pour qu'il supprime automatiquement les objets 10 jours après leur téléchargement. Toutefois, Lambda ne supprime pas automatiquement le compartiment lui-même. Pour supprimer le bucket de votre Compte AWS, suivez les instructions de la section Supprimer un bucket. Le nom du compartiment par défaut utilise le préfixe`lambdasam`, une chaîne alphanumérique à 10 chiffres, et Région AWS vous avez créé votre fonction dans :

```
lambdasam-06f22da95b-us-east-1
```

Pour éviter que des frais supplémentaires ne soient ajoutés à votre compte Compte AWS, nous vous recommandons de supprimer le compartiment Amazon S3 dès que vous avez terminé d'exporter votre fonction vers Infrastructure Composer.

La tarification standard d'Amazon S3 s'applique.

Autorisations requises

Pour utiliser la fonctionnalité d'intégration de Lambda à Infrastructure Composer, vous devez disposer de certaines autorisations pour télécharger un AWS SAM modèle et écrire la configuration de votre fonction sur Amazon S3.

Pour télécharger un AWS SAM modèle, vous devez être autorisé à utiliser les actions d'API suivantes :

- GetPolicy
- iam : GetPolicyVersion
- iam : GetRole
- iam : GetRolePolicy
- iam : ListAttachedRolePolicies
- iam : ListRolePolicies
- iam : ListRoles

Vous pouvez autoriser l'utilisation de toutes ces actions en ajoutant la politique `AWSLambda_ReadOnlyAccess` AWS gérée à votre rôle d'utilisateur IAM.

Pour que Lambda puisse écrire la configuration de votre fonction sur Amazon S3, vous devez être autorisé à utiliser les actions d'API suivantes :

- S3 : PutObject
- S3 : CreateBucket
- S3 : PutBucketEncryption
- S3 : PutBucketLifecycleConfiguration

Si vous ne pouvez pas exporter la configuration de votre fonction vers Infrastructure Composer, vérifiez que votre compte dispose des autorisations requises pour ces opérations. Si vous avez les autorisations requises mais que vous ne pouvez toujours pas exporter la configuration de votre fonction, vérifiez les politiques basées sur les ressources qui pourraient limiter l'accès Amazon S3.

Autres ressources

Pour obtenir un didacticiel plus détaillé sur la conception d'une application sans serveur dans Infrastructure Composer sur la base d'une fonction Lambda existante, consultez the section called "Infrastructure en tant que code (IaC)".

Pour utiliser Infrastructure Composer et concevoir et déployer une application sans serveur complète AWS SAM à l'aide de Lambda, vous pouvez également suivre AWS Infrastructure Composer le didacticiel du Serverless Patterns AWS Workshop.

Utilisation de AWS Lambda avec CloudFormation

Dans un modèle AWS CloudFormation, vous pouvez spécifier une fonction Lambda comme cible pour une ressource personnalisée. Utilisez des ressources personnalisées pour traiter les paramètres, récupérer des valeurs de configuration ou appeler d'autres Services AWS lors d'événements du cycle de vie de la pile.

L'exemple suivant appelle une fonction qui est définie ailleurs dans le modèle.

Example – Définition de ressource personnalisée

```
Resources:
  primerinvoke:
    Type: AWS::CloudFormation::CustomResource
    Version: "1.0"
    Properties:
      ServiceToken: !GetAtt primer.Arn
      FunctionName: !Ref randomerror
```

Le jeton de service est l'Amazon Resource Name (ARN) de la fonction appelée par CloudFormation lorsque vous créez, mettez à jour ou supprimez la pile. Vous pouvez également inclure des propriétés supplémentaires comme `FunctionName`, transmises par CloudFormation à votre fonction en l'état.

CloudFormation appelle votre fonction Lambda de façon asynchrone avec un événement qui inclut une URL de rappel.

Example –CloudFormation Événement de message

```
{
    "RequestType": "Create",
    "ServiceToken": "arn:aws:lambda:us-east-1:123456789012:function:lambda-error-
processor-primer-14ROR2T3JKU66",
    "ResponseURL": "https://cloudformation-custom-resource-response-useast1.s3-us-
east-1.amazonaws.com/arn%3Aaws%3Acloudformation%3Aus-east-1%3A123456789012%3Astack/
lambda-error-processor/1134083a-2608-1e91-9897-022501a2c456%7Cprimerinvoke
%7C5d478078-13e9-baf0-464a-7ef285ecc786?
AWSAccessKeyId=AKIAIOSFODNN7EXAMPLE&Expires=1555451971&Signature=28UijZePE5I4dvukKQqM
%2F9Rf1o4%3D",
    "StackId": "arn:aws:cloudformation:us-east-1:123456789012:stack/lambda-error-
processor/1134083a-2608-1e91-9897-022501a2c456",
    "RequestId": "5d478078-13e9-baf0-464a-7ef285ecc786",
    "LogicalResourceId": "primerinvoke",
```

```
    "ResourceType": "AWS::CloudFormation::CustomResource",
    "ResourceProperties": {
        "ServiceToken": "arn:aws:lambda:us-east-1:123456789012:function:lambda-error-
processor-primer-14ROR2T3JKU66",
        "FunctionName": "lambda-error-processor-randomerror-ZWUC391MQAJK"
    }
}
```

La fonction se charge de renvoyer à l'URL de rappel une réponse indiquant le succès ou l'échec. Pour obtenir la syntaxe de réponse complète, consultez Objets de réponse des ressources personnalisées.

Example – CloudFormation Réponse de ressource personnalisée

```
{
    "Status": "SUCCESS",
    "PhysicalResourceId": "2019/04/18/[$LATEST]b3d1bfc65f19ec610654e4d9b9de47a0",
    "StackId": "arn:aws:cloudformation:us-east-1:123456789012:stack/lambda-error-
processor/1134083a-2608-1e91-9897-022501a2c456",
    "RequestId": "5d478078-13e9-baf0-464a-7ef285ecc786",
    "LogicalResourceId": "primerinvoke"
}
```

CloudFormation fournit une bibliothèque appelée `cfn-response` qui traite l'envoi de la réponse. Si vous définissez votre fonction au sein d'un modèle, vous pouvez demander la bibliothèque par son nom. CloudFormation ajoute alors la bibliothèque au package de déploiement qu'il crée pour la fonction.

Si votre fonction qu'une ressource personnalisée utilise possède une interface réseau Elastic qui lui est associée, ajoutez les ressources suivantes à la politique VPC où **`region`** est la région dans laquelle se trouve la fonction sans les tirets. Par exemple, `us-east-1` est `useast1`. Cela permettra à la ressource personnalisée de répondre à l'URL de rappel qui envoie un signal à la pile CloudFormation.

```
arn:aws:s3:::cloudformation-custom-resource-response-region",
"arn:aws:s3:::cloudformation-custom-resource-response-region/*",
```

L'exemple de fonction suivant appelle une deuxième fonction. Si l'appel aboutit, la fonction envoie une réponse de réussite à CloudFormation et la mise à jour de la pile continue. Le modèle utilise le type de ressource AWS::Serverless::Function fourni par AWS Serverless Application Model.

Example – Fonction de ressource personnalisée

```
Transform: 'AWS::Serverless-2016-10-31'
Resources:
  primer:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs16.x
      InlineCode: |
        var aws = require('aws-sdk');
        var response = require('cfn-response');
        exports.handler = function(event, context) {
            // For Delete requests, immediately send a SUCCESS response.
            if (event.RequestType == "Delete") {
                response.send(event, context, "SUCCESS");
                return;
            }
            var responseStatus = "FAILED";
            var responseData = {};
            var functionName = event.ResourceProperties.FunctionName
            var lambda = new aws.Lambda();
            lambda.invoke({ FunctionName: functionName }, function(err, invokeResult) {
                if (err) {
                    responseData = {Error: "Invoke call failed"};
                    console.log(responseData.Error + ":\n", err);
                }
                else responseStatus = "SUCCESS";
                response.send(event, context, responseStatus, responseData);
            });
        };
      Description: Invoke a function to create a log stream.
      MemorySize: 128
      Timeout: 8
      Role: !GetAtt role.Arn
      Tracing: Active
```

Si la fonction appelée par la ressource personnalisée n'est pas définie dans un modèle, vous pouvez obtenir le code source pour `cfn-response` à partir de Module cfn-response dans le Guide de l'utilisateur AWS CloudFormation.

Pour plus d'informations sur les ressources personnalisées, consultez Ressources personnalisées dans le Guide de l'utilisateur AWS CloudFormation.

Traiter les événements Amazon DocumentDB avec Lambda

Vous pouvez utiliser une fonction Lambda pour traiter les événements dans un flux de modifications Amazon DocumentDB (compatible avec MongoDB) en configurant un cluster Amazon DocumentDB comme source d'événements. Ensuite, vous pouvez automatiser les charges de travail orientées événements en invoquant votre fonction Lambda chaque fois que les données changent avec votre cluster Amazon DocumentDB.

> **Note**
>
> Lambda prend en charge uniquement les versions 4.0 et 5.0 d'Amazon DocumentDB. Lambda ne prend pas en charge la version 3.6.
> De plus, pour les mappages des sources d'événements, Lambda ne prend en charge que les clusters basés sur des instances et les clusters régionaux. Lambda ne prend pas en charge les clusters élastiques ni les clusters globaux. Cette limitation ne s'applique pas lorsque vous utilisez Lambda en tant que client pour vous connecter à Amazon DocumentDB. Lambda peut se connecter à tous les types de clusters pour effectuer des opérations CRUD.

Lambda traite les événements des flux de modifications Amazon DocumentDB de manière séquentielle dans l'ordre de leur arrivée. Pour cette raison, votre fonction ne peut gérer qu'une seule invocation simultanée d'Amazon DocumentDB à la fois. Pour surveiller votre fonction, vous pouvez suivre ses métriques de simultanéité.

> **Warning**
>
> Les mappages des sources d'événements Lambda traitent chaque événement au moins une fois, et le traitement des enregistrements peut être dupliqué. Pour éviter les problèmes potentiels liés à des événements dupliqués, nous vous recommandons vivement de rendre votre code de fonction idempotent. Pour en savoir plus, consultez Comment rendre ma fonction Lambda idempotente dans le Knowledge Center. AWS

Rubriques

- Exemple d'événement Amazon DocumentDB
- Conditions préalables et autorisations
- Configurer la sécurité réseau

- Création d'un mappage des sources d'événements Amazon DocumentDB (console)
- Création d'un mappage des sources d'événements Amazon DocumentDB (kit SDK ou CLI)
- Positions de départ des interrogations et des flux
- Surveillance de votre source d'événements Amazon DocumentDB
- Tutoriel : Utilisation AWS Lambda avec Amazon DocumentDB Streams

Exemple d'événement Amazon DocumentDB


```
{
    "eventSourceArn": "arn:aws:rds:us-
east-1:123456789012:cluster:canaryclusterb2a659a2-qo5tcmqkcl03",
    "events": [
        {
            "event": {
                "_id": {
                    "_data": "0163eeb6e7000000090100000009000041e1"
                },
                "clusterTime": {
                    "$timestamp": {
                        "t": 1676588775,
                        "i": 9
                    }
                },
                "documentKey": {
                    "_id": {
                        "$oid": "63eeb6e7d418cd98afb1c1d7"
                    }
                },
                "fullDocument": {
                    "_id": {
                        "$oid": "63eeb6e7d418cd98afb1c1d7"
                    },
                    "anyField": "sampleValue"
                },
                "ns": {
                    "db": "test_database",
                    "coll": "test_collection"
                },
                "operationType": "insert"
            }
        }
```

```
    ],
    "eventSource": "aws:docdb"
}
```

Pour plus d'informations sur les événements de cet exemple et leurs formes, consultez la page Événements de modification sur le site Web de la documentation MongoDB.

Conditions préalables et autorisations

Avant de pouvoir utiliser Amazon DocumentDB comme source d'événements pour votre fonction Lambda, veuillez prendre note des conditions préalables suivantes. Vous devez :

- Disposez d'un cluster Amazon DocumentDB existant dans le même Compte AWS cluster Région AWS que votre fonction. Si vous n'avez pas de cluster existant, vous pouvez en créer un en suivant les étapes de la section Prise en main d'Amazon DocumentDB dans le Guide du développeur Amazon DocumentDB. Sinon, la première série d'étapes de Tutoriel : Utilisation AWS Lambda avec Amazon DocumentDB Streams vous guide dans la création d'un cluster Amazon DocumentDB avec tous les prérequis nécessaires.
- Autorisez Lambda à accéder aux ressources Amazon Virtual Private Cloud (Amazon VPC) associées à votre cluster Amazon DocumentDB. Pour de plus amples informations, veuillez consulter Configurer la sécurité réseau.
- Activez le protocole TLS sur votre cluster Amazon DocumentDB. Il s'agit du paramètre par défaut. Si vous désactivez le protocole TLS, Lambda ne peut pas communiquer avec votre cluster.
- Activez les flux de modifications sur votre cluster Amazon DocumentDB. Pour plus d'informations, veuillez consulter la rubrique Utilisation des flux de modifications avec Amazon DocumentDB dans le Guide du développeur Amazon DocumentDB.
- Fournissez à Lambda les informations d'identification pour accéder à votre cluster Amazon DocumentDB. Lors de la configuration de la source d'événement, fournissez la clé AWS Secrets Manager qui contient les informations d'authentification (nom d'utilisateur et mot de passe) requises pour accéder à votre cluster. Pour fournir cette clé lors de la configuration, procédez de l'une des manières suivantes :
  - Si vous utilisez la console Lambda pour la configuration, saisissez cette clé dans le champ Clé du gestionnaire de secrets.
  - Si vous utilisez le AWS Command Line Interface (AWS CLI) pour la configuration, fournissez cette clé dans l'`source-access-configuration`option. Vous pouvez inclure cette option avec la commande `create-event-source-mapping` ou la commande `update-event-source-mapping`. Par exemple :

```
aws lambda create-event-source-mapping \
    ...
    --source-access-configurations
 '[{"Type":"BASIC_AUTH","URI":"arn:aws:secretsmanager:us-
west-2:123456789012:secret:DocDBSecret-AbC4E6"}]' \
    ...
```

- Accordez des autorisations à Lambda pour gérer les ressources liées à votre flux Amazon DocumentDB. Ajoutez manuellement les autorisations suivantes au rôle d'exécution de votre fonction :
  - RDS : Décrivez DBClusters
  - RDS DBCluster : Décrire les paramètres
  - RDS DBSubnet : Décrire les groupes
  - EC2 : CreateNetworkInterface
  - EC2 : DescribeNetworkInterfaces
  - EC2 : DescribeVpcs
  - EC2 : DeleteNetworkInterface
  - EC2 : DescribeSubnets
  - EC2 : DescribeSecurityGroups
  - kms:Decrypt
  - responsable des secrets : GetSecretValue
- La taille des événements de flux de modifications Amazon DocumentDB que vous envoyez à Lambda doit être inférieure à 6 Mo. Lambda prend en charge des charges utiles d'une taille maximale de 6 Mo. Si votre flux de modifications essaie d'envoyer à Lambda un événement supérieur à 6 Mo, Lambda supprime le message et émet la métrique `OversizedRecordCount`. Lambda émet toutes les métriques dans la mesure du possible.

> **Note**
>
> Alors que les fonctions Lambda ont généralement un délai d'expiration maximal de 15 minutes, les mappages des sources d'événement pour Amazon MSK, Apache Kafka autogéré, Amazon DocumentDB et Amazon MQ pour ActiveMQ et RabbitMQ ne prennent en charge que les fonctions dont le délai d'expiration maximal est de 14 minutes. Cette

contrainte garantit que le mappage des sources d'événements peut gérer correctement les erreurs de fonction et effectuer de nouvelles tentatives.

Configurer la sécurité réseau

Pour donner à Lambda un accès complet à Amazon DocumentDB via votre mappage des sources d'événements, soit votre cluster doit utiliser un point de terminaison public (adresse IP publique), soit vous devez fournir un accès au VPC Amazon dans lequel vous avez créé le cluster.

Lorsque vous utilisez Amazon DocumentDB avec Lambda, créez des points de terminaison de VPC AWS PrivateLink qui permettent à votre fonction d'accéder aux ressources de votre Amazon VPC.

> **Note**
>
> AWS PrivateLink Les points de terminaison VPC sont requis pour les fonctions avec des mappages de sources d'événements qui utilisent le mode par défaut (à la demande) pour les sondeurs d'événements. Si le mappage de votre source d'événements utilise le mode provisionné, vous n'avez pas besoin de configurer les points de terminaison AWS PrivateLink VPC.

Créez un point de terminaison pour accéder aux ressources suivantes :

- Lambda — Créez un point de terminaison pour le principal de service Lambda.
- AWS STS — Créez un point de terminaison pour le AWS STS afin qu'un directeur de service assume un rôle en votre nom.
- Secrets Manager : si votre cluster utilise Secrets Manager pour stocker les informations d'identification, créez un point de terminaison pour Secrets Manager.

Vous pouvez également configurer une passerelle NAT sur chaque sous-réseau public d'Amazon VPC. Pour de plus amples informations, veuillez consulter the section called "Accès Internet pour les fonctions VPC".

Lorsque vous créez un mappage de source d'événements pour Amazon DocumentDB, Lambda vérifie si des interfaces réseau élastiques (ENIs) sont déjà présentes pour les sous-réseaux et les groupes de sécurité configurés pour votre Amazon VPC. Si Lambda trouve des objets existants ENIs,

il essaie de les réutiliser. Sinon, Lambda en crée un nouveau ENIs pour se connecter à la source de l'événement et appeler votre fonction.

> **Note**
>
> Les fonctions Lambda s'exécutent toujours au sein VPCs du service Lambda. La configuration VPC de votre fonction n'affecte pas le mappage des sources d'événements. Seule la configuration réseau des sources d'événements détermine la manière dont Lambda se connecte à votre source d'événements.

Configurez les groupes de sécurité pour le VPC Amazon contenant votre cluster. Par défaut, Amazon DocumentDB utilise les ports suivants : `27017`.

- Règles entrantes – Autorisent tout le trafic sur le port de l'agent par défaut pour le groupe de sécurité associé à votre source d'événement. Vous pouvez également utiliser une règle de groupe de sécurité avec référence circulaire pour autoriser l'accès à partir d'instances appartenant au même groupe de sécurité.
- Règles de sortie : autorisez tout le trafic sur le port `443` pour les destinations externes si votre fonction doit communiquer avec les AWS services. Vous pouvez également utiliser une règle de groupe de sécurité à référencement automatique pour limiter l'accès au courtier si vous n'avez pas besoin de communiquer avec d'autres AWS services.
- Règles entrantes relatives au point de terminaison Amazon VPC – Si vous utilisez un point de terminaison Amazon VPC, le groupe de sécurité associé à votre point de terminaison Amazon VPC doit autoriser le trafic entrant sur le port `443` en provenance du groupe de sécurité du cluster.

Si votre cluster utilise l'authentification, vous pouvez également restreindre la politique de point de terminaison pour le point de terminaison Secrets Manager. Pour appeler l'API Secrets Manager, Lambda utilise votre rôle de fonction, et non le principal de service Lambda.

Example Politique de point de terminaison de VPC — Point de terminaison Secrets Manager

```
{
      "Statement": [
          {
              "Action": "secretsmanager:GetSecretValue",
              "Effect": "Allow",
              "Principal": {
```

```
                "AWS": [
                    "arn:aws::iam::123456789012:role/my-role"
                ]
            },
            "Resource": "arn:aws::secretsmanager:us-west-2:123456789012:secret:my-
secret"
        }
    ]
}
```

Lorsque vous utilisez des points de terminaison Amazon VPC, AWS achemine vos appels d'API pour appeler votre fonction à l'aide de l'Elastic Network Interface (ENI) du point de terminaison. Le directeur du service Lambda doit faire appel à tous `lambda:InvokeFunction` les rôles et fonctions qui les utilisent. ENIs

Par défaut, les points de terminaison Amazon VPC disposent de politiques IAM ouvertes qui permettent un accès étendu aux ressources. La meilleure pratique consiste à restreindre ces politiques pour effectuer les actions nécessaires à l'aide de ce point de terminaison. Pour garantir que votre mappage des sources d'événements est en mesure d'invoquer votre fonction Lambda, la politique de point de terminaison de VPC doit autoriser le principal de service Lambda à appeler `sts:AssumeRole` et `lambda:InvokeFunction`. Le fait de restreindre vos politiques de point de terminaison de VPC pour autoriser uniquement les appels d'API provenant de votre organisation empêche le mappage des sources d'événements de fonctionner correctement. C'est pourquoi `"Resource": "*"` est requis dans ces politiques.

Les exemples de politiques de point de terminaison de VPC suivants montrent comment accorder l'accès requis au principal de service Lambda pour les points de terminaison AWS STS et Lambda.

Example Politique de point de terminaison VPC — point de terminaison AWS STS

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
```

```
            }
        ]
    }
```

Example Politique de point de terminaison de VPC — Point de terminaison Lambda

```
{
      "Statement": [
          {
              "Action": "lambda:InvokeFunction",
              "Effect": "Allow",
              "Principal": {
                  "Service": [
                      "lambda.amazonaws.com"
                  ]
              },
              "Resource": "*"
          }
      ]
  }
```

Création d'un mappage des sources d'événements Amazon DocumentDB (console)

Pour qu'une fonction Lambda puisse lire le flux de modifications d'un cluster Amazon DocumentDB, créez un mappage des sources d'événements. Cette section explique comment procéder à partir de la console Lambda. Pour le AWS SDK et les AWS CLI instructions, voirthe section called "Création d'un mappage des sources d'événements Amazon DocumentDB (kit SDK ou CLI)".

Pour créer un mappage des sources d'événements Amazon DocumentDB (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom d'une fonction.
3. Sous Function overview (Présentation de la fonction), choisissez Add trigger (Ajouter un déclencheur).
4. Sous Configuration du déclencheur, dans la liste déroulante, choisissez DocumentDB.
5. Configurez les options requises, puis choisissez Add (Ajouter).

Lambda prend en charge les options suivantes pour les sources d'événement Amazon DocumentDB :

- Cluster DocumentDB : sélectionnez un cluster Amazon DocumentDB.
- Activer le déclencheur : choisissez si vous voulez activer le déclencheur maintenant. Si vous cochez cette case, votre fonction commence immédiatement à recevoir du trafic provenant du flux de modifications Amazon DocumentDB spécifié lors de la création du mappage des sources d'événements. Nous vous recommandons de décocher la case pour créer le mappage des sources d'événements dans un état désactivé à des fins de test. Après la création, vous pouvez activer le mappage des sources d'événements à tout moment.
- Nom de la base de données – Saisissez le nom de la base de données du cluster à utiliser.
- (Facultatif) Nom de la collection : saisissez le nom d'une collection de la base de données à utiliser. Si vous n'indiquez pas de collection, Lambda écoute tous les événements de chaque collection de la base de données.
- Taille de lot – Définissez le nombre maximum de messages à extraire dans un lot, jusqu'à 10 000. La taille du lot par défaut est de 100.
- Position de départ – Choisissez la position dans le flux à partir de laquelle commencer la lecture des enregistrements.
  - Derniers – Traiter uniquement les nouveaux enregistrements qui sont ajoutés au flux. Votre fonction ne commence à traiter les enregistrements que lorsque Lambda a fini de créer votre source d'événements. Cela signifie que certains enregistrements peuvent être supprimés jusqu'à ce que la source de votre événement soit correctement créée.
  - Trim horizon (Supprimer l'horizon) – Traiter tous les enregistrements figurant dans le flux. Lambda utilise la durée de conservation des journaux de votre cluster pour déterminer par où commencer la lecture des événements. Plus précisément, Lambda commence à lire à partir de `current_time - log_retention_duration`. Votre flux de modifications doit déjà être actif avant cet horodatage pour que Lambda puisse lire correctement tous les événements.
  - At timestamp (À l'horodatage) – Traitez les enregistrements à partir d'une heure spécifique. Votre flux de modifications doit déjà être actif avant l'horodatage spécifié pour que Lambda puisse lire correctement tous les événements.
- Authentication – Choisissez la méthode d'authentification pour accéder aux agents de votre cluster.
  - BASIC_AUTH – Avec l'authentification de base, vous devez fournir la clé Secrets Manager qui contient les informations d'identification pour accéder à votre cluster.

- Clé Secrets Manager : choisissez la clé Secrets Manager qui contient les informations d'authentification (nom d'utilisateur et mot de passe) requises pour accéder à votre cluster Amazon DocumentDB.
- (Facultatif) Fenêtre de traitement par lot : définissez l'intervalle de temps maximum (en secondes) pour collecter des enregistrements avant d'invoquer votre fonction, jusqu'à 300.
- (Facultatif) Configuration complète du document : pour les opérations de mise à jour des documents, choisissez ce que vous voulez envoyer au flux. La valeur par défaut est `Default`, ce qui signifie que pour chaque événement de flux de modifications, Amazon DocumentDB envoie uniquement un delta décrivant les modifications apportées. Pour plus d'informations sur ce champ, consultez la documentation FullDocumentde l'API MongoDB Javadoc.
  - Par défaut – Lambda n'envoie qu'un document partiel décrivant les modifications apportées.
  - UpdateLookup— Lambda envoie un delta décrivant les modifications, ainsi qu'une copie de l'intégralité du document.

Création d'un mappage des sources d'événements Amazon DocumentDB (kit SDK ou CLI)

Pour créer ou gérer votre mappage des sources d'événements Amazon DocumentDB avec un kit SDK AWS, vous pouvez utiliser les opérations d'API suivantes :

- CreateEventSourceMapping
- ListEventSourceMappings
- GetEventSourceMapping
- UpdateEventSourceMapping
- DeleteEventSourceMapping

Pour créer le mappage des sources d'événements avec le AWS CLI, utilisez la `create-event-source-mapping`commande. L'exemple suivant utilise cette commande pour mapper une fonction nommée `my-function` à un flux de modifications Amazon DocumentDB. La source d'événement est spécifiée par un Amazon Resource Name (ARN), avec une taille de lot de 500, à partir de l'horodatage en heure Unix. La commande spécifie également la clé Secrets Manager que Lambda utilise pour se connecter à Amazon DocumentDB. De plus, elle inclut des paramètres `document-db-event-source-config` qui spécifient la base de données et la collection à partir de laquelle lire.

```
aws lambda create-event-source-mapping --function-name my-function \
    --event-source-arn arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy
    --batch-size 500 \
    --starting-position AT_TIMESTAMP \
    --starting-position-timestamp 1541139109 \
    --source-access-configurations
 '[{"Type":"BASIC_AUTH","URI":"arn:aws:secretsmanager:us-
east-1:123456789012:secret:DocDBSecret-BAtjxi"}]' \
    --document-db-event-source-config '{"DatabaseName":"test_database",
 "CollectionName": "test_collection"}' \
```

Vous devriez obtenir un résultat du type suivant :

```
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
    "BatchSize": 500,
    "DocumentDBEventSourceConfig": {
        "CollectionName": "test_collection",
        "DatabaseName": "test_database",
        "FullDocument": "Default"
    },
    "MaximumBatchingWindowInSeconds": 0,
    "EventSourceArn": "arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "LastModified": 1541348195.412,
    "LastProcessingResult": "No records processed",
    "State": "Creating",
    "StateTransitionReason": "User action"
}
```

Après la création, vous pouvez utiliser la commande `update-event-source-mapping` pour mettre à jour les paramètres de votre source d'événements Amazon DocumentDB. L'exemple suivant met à jour la taille du lot à 1 000 et la fenêtre de traitement par lots à 10 secondes. Pour cette commande, vous avez besoin de l'UUID de votre mappage des sources d'événements, que vous pouvez récupérer à l'aide de la commande `list-event-source-mapping` ou de la console Lambda.

```
aws lambda update-event-source-mapping --function-name my-function \
    --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
```

```
    --batch-size 1000 \
    --batch-window 10
```

Vous devriez obtenir un résultat du type suivant :

```
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
    "BatchSize": 500,
    "DocumentDBEventSourceConfig": {
        "CollectionName": "test_collection",
        "DatabaseName": "test_database",
        "FullDocument": "Default"
    },
    "MaximumBatchingWindowInSeconds": 0,
    "EventSourceArn": "arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "LastModified": 1541359182.919,
    "LastProcessingResult": "OK",
    "State": "Updating",
    "StateTransitionReason": "User action"
}
```

Lambda met à jour les paramètres de façon asynchrone, il se peut donc que vous ne voyiez pas ces modifications dans la sortie tant que le processus n'est pas terminé. Pour afficher les paramètres actuels de votre mappage des sources d'événements, utilisez la commande `get-event-source-mapping`.

```
aws lambda get-event-source-mapping --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b
```

Vous devriez obtenir un résultat du type suivant :

```
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
    "DocumentDBEventSourceConfig": {
        "CollectionName": "test_collection",
        "DatabaseName": "test_database",
        "FullDocument": "Default"
    },
    "BatchSize": 1000,
    "MaximumBatchingWindowInSeconds": 10,
```

```
    "EventSourceArn": "arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "LastModified": 1541359182.919,
    "LastProcessingResult": "OK",
    "State": "Enabled",
    "StateTransitionReason": "User action"
}
```

Pour supprimer le mappage des sources d'événements Amazon DocumentDB, utilisez la commande [delete-event-source-mapping](#).

```
aws lambda delete-event-source-mapping \
    --uuid 2b733gdc-8ac3-cdf5-af3a-1827b3b11284
```

Positions de départ des interrogations et des flux

Sachez que l'interrogation des flux lors des mises à jour et de la création du mappage des sources d'événements est finalement cohérente.

- Lors de la création du mappage des sources d'événements, le démarrage de l'interrogation des événements depuis le flux peut prendre plusieurs minutes.
- Lors des mises à jour du mappage des sources d'événements, l'arrêt et le redémarrage de l'interrogation des événements depuis le flux peuvent prendre plusieurs minutes.

Ce comportement signifie que si vous spécifiez `LATEST` comme position de départ du flux, le mappage des sources d'événements peut manquer des événements lors de la création ou des mises à jour. Pour vous assurer de ne manquer aucun événement, spécifiez la position de départ du flux comme `TRIM_HORIZON` ou `AT_TIMESTAMP`.

Surveillance de votre source d'événements Amazon DocumentDB

Pour vous aider à surveiller votre source d'événements Amazon DocumentDB, Lambda émet la métrique `IteratorAge` lorsque votre fonction termine le traitement d'un lot d'enregistrements. L'âge de l'itérateur est la différence entre l'horodatage de l'événement le plus récent et l'horodatage actuel. La métrique `IteratorAge` indique essentiellement l'ancienneté du dernier enregistrement traité dans le lot. Si votre fonction traite actuellement de nouveaux événements, vous pouvez utiliser l'âge de l'itérateur pour estimer la latence entre le moment où un enregistrement est ajouté et celui où

votre fonction le traite. Une tendance à la hausse de `IteratorAge` peut indiquer des problèmes liés à votre fonction. Pour de plus amples informations, veuillez consulter Utilisation des métriques CloudWatch avec Lambda.

Les flux de modifications Amazon DocumentDB ne sont pas optimisés pour gérer les intervalles de temps importants entre les événements. Si votre source d'événements Amazon DocumentDB ne reçoit aucun événement pendant une longue période, Lambda peut désactiver le mappage des sources d'événements. Cette période peut varier de quelques semaines à quelques mois en fonction de la taille du cluster et des autres charges de travail.

Lambda prend en charge des charges utiles allant jusqu'à 6 Mo. Cependant, les événements du flux de modification d'Amazon DocumentDB peuvent avoir une taille allant jusqu'à 16 Mo. Si votre flux de modifications tente d'envoyer à Lambda un événement d'une taille supérieure à 6 Mo, Lambda supprime le message et émet la métrique `OversizedRecordCount`. Lambda émet toutes les métriques dans la mesure du possible.

Tutoriel : Utilisation AWS Lambda avec Amazon DocumentDB Streams

Dans ce tutoriel, vous créez une fonction Lambda de base qui consomme des événements à partir d'un flux de modifications Amazon DocumentDB (compatible avec MongoDB). Pour réaliser ce tutoriel, vous passerez par les étapes suivantes :

- Configurez votre cluster Amazon DocumentDB, connectez-vous-y, et activez les flux de modifications sur ce cluster.
- Créez votre fonction Lambda, et configurez votre cluster Amazon DocumentDB en tant que source d'événements pour votre fonction.
- Testez la configuration en insérant des éléments dans votre base de données Amazon DocumentDB.

Créer le cluster Amazon DocumentDB

1. Ouvrez la console Amazon DocumentDB. Sous Clusters, sélectionnez Créer.
2. Créez un cluster avec la configuration suivante :

    - Pour Type de cluster, choisissez Cluster basé sur l'instance. Il s'agit de l'option par défaut.
    - Sous Configuration du cluster, assurez-vous que la Version du moteur 5.0.0 est sélectionnée. Il s'agit de l'option par défaut.
    - Sous Configuration de l'instance :

    - Pour Classe d’instance de base de données, sélectionnez Classes à mémoire optimisée. Il s’agit de l’option par défaut.
    - Pour Nombre d’instances de réplica ordinaire, choisissez 1.
    - Pour Classe de l’instance, utilisez la sélection par défaut.
  - Sous Authentification, saisissez un nom d’utilisateur pour l’utilisateur principal, puis choisissez Autogéré. Saisissez un mot de passe, puis confirmez-le.
  - Conservez tous les autres paramètres par défaut.
3. Choisissez Créer un cluster.

Création d’un secret dans Secrets Manager

Pendant qu'Amazon DocumentDB crée votre cluster, créez un AWS Secrets Manager secret pour stocker les informations d'identification de votre base de données. Vous fournirez ce secret lors de la création du mappage des sources d’événements Lambda lors d’une étape ultérieure.

Pour créer le secret dans Secrets Manager

1. Ouvrez la console Secrets Manager et choisissez Stocker un nouveau secret.
2. Pour Choisir le type de secret, sélectionnez les options suivantes :
    - Sous Informations de base :
      - Type de secret : informations d’identification pour votre base de données Amazon DocumentDB
      - Sous Informations d’identification, saisissez les mêmes nom d’utilisateur et mot de passe que vous avez utilisés pour créer votre cluster Amazon DocumentDB.
      - Base de données : choisissez votre cluster Amazon DocumentDB.
      - Choisissez Suivant.
3. Pour Configurer le secret, choisissez les options suivantes :
    - Nom secret : `DocumentDBSecret`
    - Choisissez Suivant.
4. Choisissez Suivant.
5. Choisissez Stocker.
6. Actualisez la console pour vérifier que vous avez correctement enregistré le secret `DocumentDBSecret`.

Notez l'ARN du secret. Vous en aurez besoin dans une étape ultérieure.

Connexion au cluster

Connectez-vous à votre cluster Amazon DocumentDB à l'aide de AWS CloudShell

1. Dans la console de gestion Amazon DocumentDB, sous Clusters, recherchez le cluster que vous avez créé. Sélectionnez votre cluster en cochant la case en regard de celui-ci.
2. Choisissez Se connecter au cluster. L'écran CloudShell Exécuter la commande apparaît.
3. Dans le champ Nom du nouvel environnement, saisissez un nom unique, comme « test », puis choisissez Créer et exécuter.
4. Lorsque vous y êtes invité, saisissez votre mot de passe. Lorsque l'invite affiche `rs0 [direct: primary] <env-name>>`, vous êtes connecté avec succès à votre cluster Amazon DocumentDB.

Activation des flux de modifications

Pour ce tutoriel, vous allez suivre les modifications apportées à la collection `products` de la base de données `docdbdemo` dans votre cluster Amazon DocumentDB. Pour ce faire, vous activez les flux de modifications.

Pour créer une nouvelle base de données dans votre cluster

1. Exécutez la commande suivante pour créer une nouvelle base de données appelée `docdbdemo` :

   ```
   use docdbdemo
   ```

2. Dans la fenêtre de terminal, utilisez la commande suivante pour insérer un enregistrement dans `docdbdemo` :

   ```
   db.products.insertOne({"hello":"world"})
   ```

   Elle doit générer une sortie comme suit :

   ```
   {
     acknowledged: true,
     insertedId: ObjectId('67f85066ca526410fd531d59')
   ```

```
}
```

3. Activez ensuite les flux de modifications sur la collection `products` de la base de données `docdbdemo` à l'aide de la commande suivante :

```
db.adminCommand({modifyChangeStreams: 1,
    database: "docdbdemo",
    collection: "products",
    enable: true});
```

Vous devriez obtenir un résultat du type suivant :

```
{ "ok" : 1, "operationTime" : Timestamp(1680126165, 1) }
```

Création de points de terminaison d'un VPC d'interface

Créez ensuite des points de terminaison d'un VPC d'interface pour vous assurer que Lambda et Secrets Manager (utilisé plus tard pour stocker nos informations d'identification d'accès au cluster) peuvent se connecter à votre VPC par défaut.

Pour créer des points de terminaison d'un VPC d'interface

1. Ouvrez la console VPC. Dans le menu de gauche, sous Cloud privé virtuel, choisissez Points de terminaison.
2. Choisissez Créer un point de terminaison. Créez un point de terminaison avec la configuration suivante :
   - Pour Balise de nom, saisissez `lambda-default-vpc`.
   - Pour la catégorie de service, sélectionnez AWS services.
   - Pour Services, saisissez `lambda` dans la zone de recherche. Choisissez le service au format `com.amazonaws.<region>.lambda`.
   - Pour VPC, choisissez le VPC dans lequel se trouve votre cluster Amazon DocumentDB. Il s'agit généralement du VPC par défaut.
   - Pour Sous-réseaux, cochez les cases à côté de chaque zone de disponibilité. Choisissez l'ID de sous-réseau correct pour chaque zone de disponibilité.
   - Pour le type d'adresse IP, sélectionnez IPv4.

- Pour Groupes de sécurité, choisissez le groupe de sécurité utilisé par votre cluster Amazon DocumentDB. Il s'agit généralement du groupe de sécurité `default`.
- Conservez tous les autres paramètres par défaut.
- Choisissez Créer un point de terminaison.

3. Choisissez à nouveau Créer un point de terminaison. Créez un point de terminaison avec la configuration suivante :

   - Pour Balise de nom, saisissez `secretsmanager-default-vpc`.
   - Pour la catégorie de service, sélectionnez AWS services.
   - Pour Services, saisissez `secretsmanager` dans la zone de recherche. Choisissez le service au format `com.amazonaws.<region>.secretsmanager`.
   - Pour VPC, choisissez le VPC dans lequel se trouve votre cluster Amazon DocumentDB. Il s'agit généralement du VPC par défaut.
   - Pour Sous-réseaux, cochez les cases à côté de chaque zone de disponibilité. Choisissez l'ID de sous-réseau correct pour chaque zone de disponibilité.
   - Pour le type d'adresse IP, sélectionnez IPv4.
   - Pour Groupes de sécurité, choisissez le groupe de sécurité utilisé par votre cluster Amazon DocumentDB. Il s'agit généralement du groupe de sécurité `default`.
   - Conservez tous les autres paramètres par défaut.
   - Choisissez Créer un point de terminaison.

Ceci termine la partie de ce tutoriel concernant la configuration du cluster.

Créer le rôle d'exécution

Dans les étapes suivantes, vous allez créer votre fonction Lambda. Tout d'abord, vous devez créer le rôle d'exécution qui donne à votre fonction l'autorisation d'accéder à votre cluster. Vous faites cela en créant d'abord une politique IAM, puis en associant cette politique à un rôle IAM.

Pour créer une politique IAM

1. Ouvrez la page Politiques dans la console IAM et choisissez Créer une politique.
2. Choisissez l'onglet JSON. Dans la politique suivante, remplacez l'ARN de la ressource Secrets Manager dans la dernière ligne de l'instruction par votre ARN secret précédent et copiez la politique dans l'éditeur.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "LambdaESMNetworkingAccess",
            "Effect": "Allow",
            "Action": [
                "ec2:CreateNetworkInterface",
                "ec2:DescribeNetworkInterfaces",
                "ec2:DescribeVpcs",
                "ec2:DeleteNetworkInterface",
                "ec2:DescribeSubnets",
                "ec2:DescribeSecurityGroups",
                "kms:Decrypt"
            ],
            "Resource": "*"
        },
        {
            "Sid": "LambdaDocDBESMAccess",
            "Effect": "Allow",
            "Action": [
                "rds:DescribeDBClusters",
                "rds:DescribeDBClusterParameters",
                "rds:DescribeDBSubnetGroups"
            ],
            "Resource": "*"
        },
        {
            "Sid": "LambdaDocDBESMGetSecretValueAccess",
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue"
            ],
            "Resource": "arn:aws:secretsmanager:us-
east-1:123456789012:secret:DocumentDBSecret"
        }
    ]
}
```

3. Choisissez Suivant : Balises, puis Suivant : Vérification.

4. Pour Nom, saisissez `AWSDocumentDBLambdaPolicy`.
5. Sélectionnez Create policy (Créer une politique).

Pour créer le rôle IAM

1. Ouvrez la page Rôles dans la console IAM et choisissez Créer un rôle.
2. Pour Sélectionner une entité de confiance, choisissez les options suivantes :
   - Type d'entité de confiance : AWS service
   - Service ou cas d’utilisation : Lambda
   - Choisissez Suivant.
3. Pour Ajouter des autorisations, choisissez la `AWSDocumentDBLambdaPolicy` politique que vous venez de créer, ainsi que celle permettant `AWSLambdaBasicExecutionRole` à votre fonction d'écrire sur Amazon CloudWatch Logs.
4. Choisissez Suivant.
5. Pour le Nom du rôle, saisissez `AWSDocumentDBLambdaExecutionRole`.
6. Choisissez Créer un rôle.

Créer la fonction Lambda

Ce didacticiel utilise le moteur d'exécution Python 3.14, mais nous avons également fourni des exemples de fichiers de code pour d'autres environnements d'exécution. Vous pouvez sélectionner l’onglet dans la zone suivante pour voir le code d’exécution qui vous intéresse.

Le code reçoit une entrée d’événement Amazon DocumentDB et traite le message qu’elle contient.

Pour créer la fonction Lambda

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez Créer une fonction.
3. Choisissez Créer à partir de zéro.
4. Sous Basic information (Informations de base), procédez comme suit :
   a. Sous Nom de la fonction, saisissez `ProcessDocumentDBRecords`.
   b. Pour Runtime, choisissez Python 3.14.

c. Pour Architecture, choisissez x86_64.

5. Dans l'onglet Modifier le rôle d'exécution par défaut, procédez comme suit :

   a. Ouvrez l'onglet, puis choisissez Utiliser un rôle existant.

   b. Sélectionnez le `AWSDocumentDBLambdaExecutionRole` que vous avez créé précédemment.

6. Choisissez Créer une fonction.

Pour déployer le code de la fonction

1. Choisissez l'onglet Python dans la zone suivante et copiez le code.

   .NET

   SDK pour .NET

   > **Note**
   >
   > Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

   Consommation d'un événement Amazon DocumentDB avec Lambda en utilisant .NET.

```
using Amazon.Lambda.Core;
using System.Text.Json;
using System;
using System.Collections.Generic;
using System.Text.Json.Serialization;
//Assembly attribute to enable the Lambda function's JSON input to be
 converted into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJson

namespace LambdaDocDb;

public class Function
{

     /// <summary>
```

```
    /// Lambda function entry point to process Amazon DocumentDB events.
    /// </summary>
    /// <param name="event">The Amazon DocumentDB event.</param>
    /// <param name="context">The Lambda context object.</param>
    /// <returns>A string to indicate successful processing.</returns>
    public string FunctionHandler(Event evnt, ILambdaContext context)
    {

        foreach (var record in evnt.Events)
        {
            ProcessDocumentDBEvent(record, context);
        }

        return "OK";
    }

     private void ProcessDocumentDBEvent(DocumentDBEventRecord record,
ILambdaContext context)
    {

        var eventData = record.Event;
        var operationType = eventData.OperationType;
        var databaseName = eventData.Ns.Db;
        var collectionName = eventData.Ns.Coll;
        var fullDocument = JsonSerializer.Serialize(eventData.FullDocument,
new JsonSerializerOptions { WriteIndented = true });

        context.Logger.LogLine($"Operation type: {operationType}");
        context.Logger.LogLine($"Database: {databaseName}");
        context.Logger.LogLine($"Collection: {collectionName}");
        context.Logger.LogLine($"Full document:\n{fullDocument}");
    }



    public class Event
    {
        [JsonPropertyName("eventSourceArn")]
        public string EventSourceArn { get; set; }

        [JsonPropertyName("events")]
        public List<DocumentDBEventRecord> Events { get; set; }

        [JsonPropertyName("eventSource")]
```

```
        public string EventSource { get; set; }
    }

    public class DocumentDBEventRecord
    {
        [JsonPropertyName("event")]
        public EventData Event { get; set; }
    }

    public class EventData
    {
        [JsonPropertyName("_id")]
        public IdData Id { get; set; }

        [JsonPropertyName("clusterTime")]
        public ClusterTime ClusterTime { get; set; }

        [JsonPropertyName("documentKey")]
        public DocumentKey DocumentKey { get; set; }

        [JsonPropertyName("fullDocument")]
        public Dictionary<string, object> FullDocument { get; set; }

        [JsonPropertyName("ns")]
        public Namespace Ns { get; set; }

        [JsonPropertyName("operationType")]
        public string OperationType { get; set; }
    }

    public class IdData
    {
        [JsonPropertyName("_data")]
        public string Data { get; set; }
    }

    public class ClusterTime
    {
        [JsonPropertyName("$timestamp")]
        public Timestamp Timestamp { get; set; }
    }

    public class Timestamp
    {
```

```
        [JsonPropertyName("t")]
        public long T { get; set; }

        [JsonPropertyName("i")]
        public int I { get; set; }
    }

    public class DocumentKey
    {
        [JsonPropertyName("_id")]
        public Id Id { get; set; }
    }

    public class Id
    {
        [JsonPropertyName("$oid")]
        public string Oid { get; set; }
    }

    public class Namespace
    {
        [JsonPropertyName("db")]
        public string Db { get; set; }

        [JsonPropertyName("coll")]
        public string Coll { get; set; }
    }
}
```

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon DocumentDB avec Lambda en utilisant Go.

```
package main

import (
 "context"
 "encoding/json"
 "fmt"

 "github.com/aws/aws-lambda-go/lambda"
)

type Event struct {
 Events []Record `json:"events"`
}

type Record struct {
 Event struct {
  OperationType string `json:"operationType"`
  NS            struct {
   DB   string `json:"db"`
   Coll string `json:"coll"`
  } `json:"ns"`
  FullDocument interface{} `json:"fullDocument"`
 } `json:"event"`
}

func main() {
 lambda.Start(handler)
}

func handler(ctx context.Context, event Event) (string, error) {
 fmt.Println("Loading function")
 for _, record := range event.Events {
  logDocumentDBEvent(record)
 }

 return "OK", nil
}

func logDocumentDBEvent(record Record) {
 fmt.Printf("Operation type: %s\n", record.Event.OperationType)
 fmt.Printf("db: %s\n", record.Event.NS.DB)
 fmt.Printf("collection: %s\n", record.Event.NS.Coll)
```

```
 docBytes, _ := json.MarshalIndent(record.Event.FullDocument, "", "  ")
 fmt.Printf("Full document: %s\n", string(docBytes))
}
```

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon DocumentDB avec Lambda en utilisant Java.

```
import java.util.List;
import java.util.Map;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

public class Example implements RequestHandler<Map<String, Object>, String> {

    @SuppressWarnings("unchecked")
    @Override
    public String handleRequest(Map<String, Object> event, Context context) {
        List<Map<String, Object>> events = (List<Map<String, Object>>)
 event.get("events");
        for (Map<String, Object> record : events) {
            Map<String, Object> eventData = (Map<String, Object>)
 record.get("event");
            processEventData(eventData);
        }

        return "OK";
    }

    @SuppressWarnings("unchecked")
    private void processEventData(Map<String, Object> eventData) {
```

```
        String operationType = (String) eventData.get("operationType");
        System.out.println("operationType: %s".formatted(operationType));

        Map<String, Object> ns = (Map<String, Object>) eventData.get("ns");

        String db = (String) ns.get("db");
        System.out.println("db: %s".formatted(db));
        String coll = (String) ns.get("coll");
        System.out.println("coll: %s".formatted(coll));

        Map<String, Object> fullDocument = (Map<String, Object>)
 eventData.get("fullDocument");
        System.out.println("fullDocument: %s".formatted(fullDocument));
    }

}
```

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon DocumentDB avec Lambda à l'aide de. JavaScript

```
console.log('Loading function');
exports.handler = async (event, context) => {
    event.events.forEach(record => {
        logDocumentDBEvent(record);
    });
    return 'OK';
};

const logDocumentDBEvent = (record) => {
    console.log('Operation type: ' + record.event.operationType);
```

```
    console.log('db: ' + record.event.ns.db);
    console.log('collection: ' + record.event.ns.coll);
    console.log('Full document:', JSON.stringify(record.event.fullDocument,
 null, 2));
};
```

Utilisation d'un événement Amazon DocumentDB avec Lambda à l'aide de TypeScript

```
import { DocumentDBEventRecord, DocumentDBEventSubscriptionContext } from
 'aws-lambda';

console.log('Loading function');

export const handler = async (
  event: DocumentDBEventSubscriptionContext,
  context: any
): Promise<string> => {
  event.events.forEach((record: DocumentDBEventRecord) => {
    logDocumentDBEvent(record);
  });
  return 'OK';
};

const logDocumentDBEvent = (record: DocumentDBEventRecord): void => {
  console.log('Operation type: ' + record.event.operationType);
  console.log('db: ' + record.event.ns.db);
  console.log('collection: ' + record.event.ns.coll);
  console.log('Full document:', JSON.stringify(record.event.fullDocument,
 null, 2));
};
```

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon DocumentDB avec Lambda en utilisant PHP.

```
<?php

require __DIR__.'/vendor/autoload.php';

use Bref\Context\Context;
use Bref\Event\Handler;

class DocumentDBEventHandler implements Handler
{
    public function handle($event, Context $context): string
    {

        $events = $event['events'] ?? [];
        foreach ($events as $record) {
            $this->logDocumentDBEvent($record['event']);
        }
        return 'OK';
    }

    private function logDocumentDBEvent($event): void
    {
        // Extract information from the event record

        $operationType = $event['operationType'] ?? 'Unknown';
        $db = $event['ns']['db'] ?? 'Unknown';
        $collection = $event['ns']['coll'] ?? 'Unknown';
        $fullDocument = $event['fullDocument'] ?? [];

        // Log the event details
```

```
        echo "Operation type: $operationType\n";
        echo "Database: $db\n";
        echo "Collection: $collection\n";
        echo "Full document: " . json_encode($fullDocument,
 JSON_PRETTY_PRINT) . "\n";
    }
}
return new DocumentDBEventHandler();
```

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon DocumentDB avec Lambda en utilisant Python.

```
import json

def lambda_handler(event, context):
    for record in event.get('events', []):
        log_document_db_event(record)
    return 'OK'

def log_document_db_event(record):
    event_data = record.get('event', {})
    operation_type = event_data.get('operationType', 'Unknown')
    db = event_data.get('ns', {}).get('db', 'Unknown')
    collection = event_data.get('ns', {}).get('coll', 'Unknown')
    full_document = event_data.get('fullDocument', {})

    print(f"Operation type: {operation_type}")
    print(f"db: {db}")
    print(f"collection: {collection}")
    print("Full document:", json.dumps(full_document, indent=2))
```

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon DocumentDB avec Lambda en utilisant Ruby.

```
require 'json'

def lambda_handler(event:, context:)
  event['events'].each do |record|
    log_document_db_event(record)
  end
  'OK'
end

def log_document_db_event(record)
  event_data = record['event'] || {}
  operation_type = event_data['operationType'] || 'Unknown'
  db = event_data.dig('ns', 'db') || 'Unknown'
  collection = event_data.dig('ns', 'coll') || 'Unknown'
  full_document = event_data['fullDocument'] || {}

  puts "Operation type: #{operation_type}"
  puts "db: #{db}"
  puts "collection: #{collection}"
  puts "Full document: #{JSON.pretty_generate(full_document)}"
end
```

Rust

SDK pour Rust

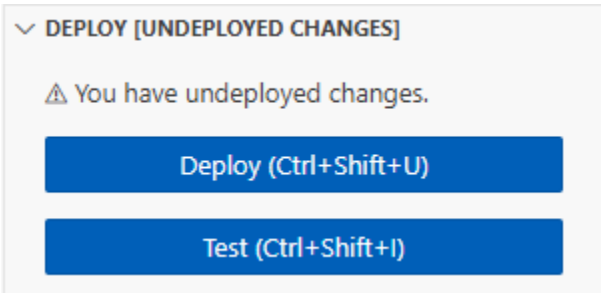
> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon DocumentDB avec Lambda en utilisant Rust.

```
use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::documentdb::{DocumentDbEvent, DocumentDbInnerEvent},
   };


// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
//tracing-subscriber = { version = "0.3", default-features = false, features
 = ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<DocumentDbEvent>) ->Result<(),
 Error> {

    tracing::info!("Event Source ARN: {:?}", event.payload.event_source_arn);
    tracing::info!("Event Source: {:?}", event.payload.event_source);

    let records = &event.payload.events;

    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }
```

```
    for record in records{
        log_document_db_event(record);
    }

    tracing::info!("Document db records processed");

    // Prepare the response
    Ok(())

}

fn log_document_db_event(record: &DocumentDbInnerEvent)-> Result<(), Error>{
    tracing::info!("Change Event: {:?}", record.event);

    Ok(())

}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())

}
```

2. Dans le volet Code source de la console Lambda, collez le code dans l'éditeur de code, en remplaçant le code créé par Lambda.
3. Dans la section DÉPLOYER, choisissez Déployer pour mettre à jour le code de votre fonction :



Création du mappage des sources d'événements Lambda

Créez le mappage des sources d'événements qui associe votre flux de modifications Amazon DocumentDB à votre fonction Lambda. Une fois que vous avez créé ce mappage des sources d'événements, commence AWS Lambda immédiatement à interroger le flux.

Pour créer le mappage des sources d'événements

1. Ouvrez la page Fonctions de la console Lambda.
2. Choisissez la fonction `ProcessDocumentDBRecords` que vous avez créée précédemment.
3. Choisissez l'onglet Configuration, puis Déclencheurs dans le menu de gauche.
4. Choisissez Add trigger (Ajouter déclencheur).
5. Sous Configuration du déclencheur, pour la source, sélectionnez Amazon DocumentDB.
6. Créez le mappage des sources d'événements avec la configuration suivante :
   - Cluster Amazon DocumentDB : choisissez le cluster que vous avez créé précédemment.
   - Nom de la base de données : docdbdemo
   - Nom de la collection : products
   - Taille du lot : 1
   - Position de départ : dernière
   - Authentification : BASIC_AUTH
   - Clé Secrets Manager : choisissez le secret de votre cluster Amazon DocumentDB. Son nom ressemblera à `rds!cluster-12345678-a6f0-52c0-b290-db4aga89274f`.
   - Fenêtre de traitement par lots : 1
   - Configuration complète du document : UpdateLookup
7. Choisissez Ajouter. La création de votre mappage des sources d'événements peut prendre quelques minutes.

Test de votre fonction

Attendez que le mappage des sources d'événements atteigne l'état Activé. Cela peut prendre plusieurs minutes. Testez ensuite la end-to-end configuration en insérant, en mettant à jour et en supprimant des enregistrements de base de données. Avant de commencer :

1. Reconnectez-vous à votre cluster Amazon DocumentDB dans CloudShell votre environnement.
2. Exécutez la commande suivante pour vous assurer que vous utilisez la base de données `docdbdemo` :

```
use docdbdemo
```

Insérer un enregistrement

Insérez un enregistrement dans la collection `products` de la base de donnée `docdbdemo` :

```
db.products.insertOne({"name":"Pencil", "price": 1.00})
```

Vérifiez que votre fonction a correctement traité cet événement en vérifiant CloudWatch les journaux. Vous devriez voir une entrée de journal comme celle-ci :

Timestamp	Message
2025-05-05T23:55:23.474Z	Operation type: insert
2025-05-05T23:55:23.474Z	db: docdbdemo
2025-05-05T23:55:23.474Z	collection: products
2025-05-05T23:55:23.474Z	Full document: {
2025-05-05T23:55:23.474Z	"_id": {
2025-05-05T23:55:23.474Z	"$oid": "68194fea08d12462ea531d58"
2025-05-05T23:55:23.474Z	},
2025-05-05T23:55:23.474Z	"name": "Pencil",
2025-05-05T23:55:23.474Z	"price": 1
2025-05-05T23:55:23.474Z	}

Mettre à jour un enregistrement

Mettez à jour l'enregistrement que vous venez d'insérer à l'aide de la commande suivante :

```
db.products.updateOne(
    { "name": "Pencil" },
    { $set: { "price": 0.50 }}
)
```

Vérifiez que votre fonction a correctement traité cet événement en vérifiant CloudWatch les journaux. Vous devriez voir une entrée de journal comme celle-ci :

Timestamp	Message
2025-05-05T23:55:27.918Z	Operation type: update
2025-05-05T23:55:27.918Z	db: docdbdemo
2025-05-05T23:55:27.918Z	collection: products
2025-05-05T23:55:27.918Z	Full document: {
2025-05-05T23:55:27.918Z	"_id": {
2025-05-05T23:55:27.918Z	"$oid": "68194fea08d12462ea531d58"
2025-05-05T23:55:27.918Z	},
2025-05-05T23:55:27.918Z	"name": "Pencil",
2025-05-05T23:55:27.918Z	"price": 0.5
2025-05-05T23:55:27.918Z	}

Supprimer un enregistrement

Supprimez l'enregistrement que vous venez de mettre à jour à l'aide de la commande suivante :

```
db.products.deleteOne( { "name": "Pencil" } )
```

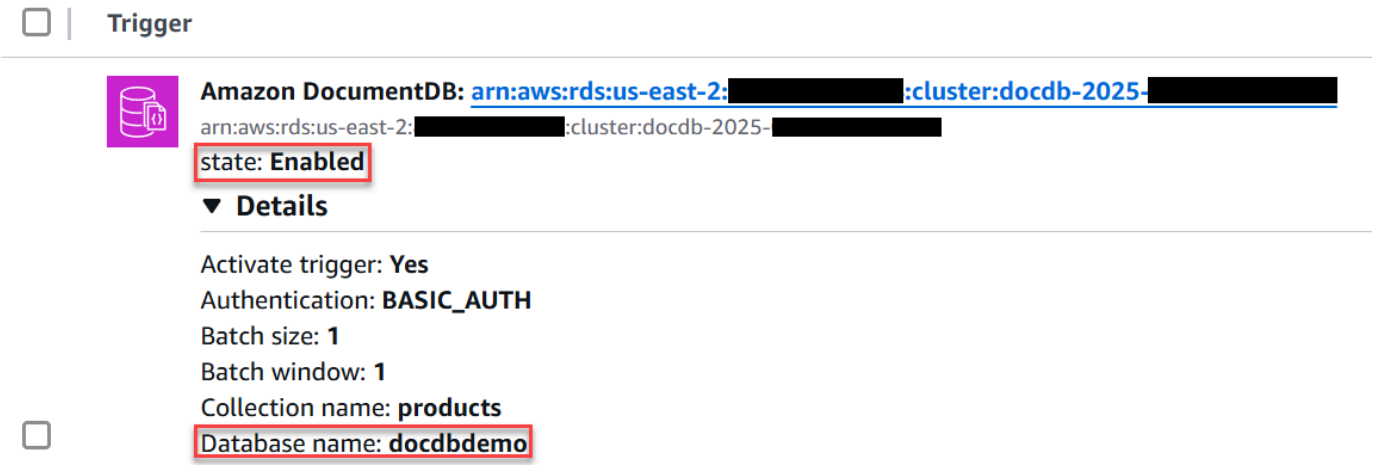
Vérifiez que votre fonction a correctement traité cet événement en vérifiant CloudWatch les journaux. Vous devriez voir une entrée de journal comme celle-ci :

Timestamp	Message
2025-05-05T23:55:32.362Z	Operation type: delete
2025-05-05T23:55:32.362Z	db: docdbdemo
2025-05-05T23:55:32.362Z	collection: products
2025-05-05T23:55:32.362Z	Full document: {}

Résolution des problèmes

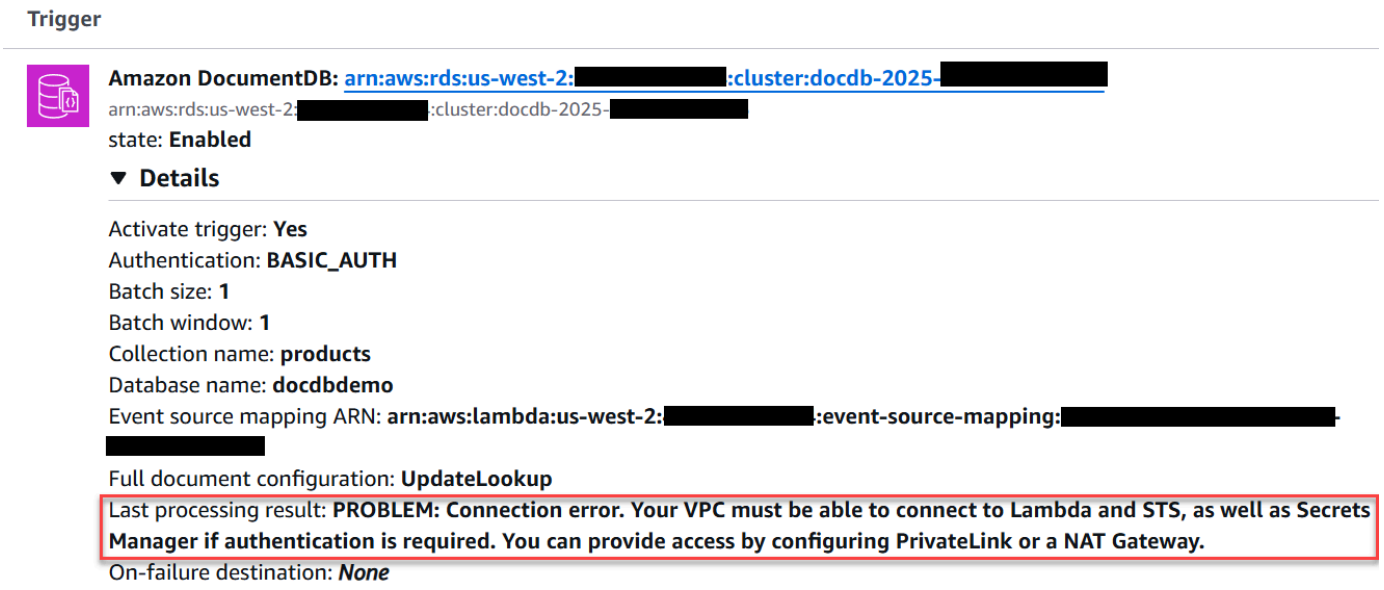
Si aucun événement de base de données n'apparaît dans les CloudWatch journaux de votre fonction, vérifiez les points suivants :

- Assurez-vous que le mappage des sources d'événements Lambda (également appelé déclencheur) est à l'état Activé. La création de mappages des sources d'événements peut prendre plusieurs minutes.
- Si le mappage des sources d'événements est activé mais que les événements de base de données ne s'affichent toujours pas dans CloudWatch :
  - Assurez-vous que le Nom de la base de données dans le mappage des sources d'événements est défini sur `docdbdemo`.



  - Vérifiez si le champ Résultat du dernier traitement du mappage des sources d'événements contient le message « PROBLEM: Connection error. Your VPC must be able to connect to Lambda and STS, as well as Secrets Manager if authentication is required. » Si cette erreur s'affiche, assurez-vous que vous avez créé les points de terminaison de l'interface VPC de

Lambda et Secrets Manager, et que les points de terminaison utilisent le même VPC et les mêmes sous-réseaux que ceux utilisés par votre cluster Amazon DocumentDB.



Nettoyage de vos ressources

Vous pouvez maintenant supprimer les ressources que vous avez créées pour ce didacticiel, sauf si vous souhaitez les conserver. En supprimant des ressources AWS que vous n'utilisez plus, vous évitez les frais superflus pour votre Compte AWS.

Pour supprimer la fonction Lambda

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction que vous avez créée.
3. Sélectionnez Actions, Supprimer.
4. Saisissez **`confirm`** dans la zone de saisie de texte et choisissez Delete (Supprimer).

Pour supprimer le rôle d'exécution

1. Ouvrez la page Roles (Rôles) de la console IAM.
2. Sélectionnez le rôle d'exécution que vous avez créé.
3. Sélectionnez Delete (Supprimer).
4. Saisissez le nom du rôle dans le champ de saisie de texte et choisissez Delete (Supprimer).

Pour supprimer les points de terminaison d'un VPC

1. Ouvrez la console VPC. Dans le menu de gauche, sous Cloud privé virtuel, choisissez Points de terminaison.
2. Sélectionnez les points de terminaison que vous avez créés.
3. Choisissez Actions, Delete VPC endpoints (Supprimer le point de terminaison d'un VPC).
4. Saisissez **`delete`** dans le champ de saisie de texte.
5. Sélectionnez Delete (Supprimer).

Pour supprimer le cluster Amazon DocumentDB

1. Ouvrez la console Amazon DocumentDB.
2. Sélectionnez le cluster Amazon DocumentDB que vous avez créé pour ce tutoriel et désactivez la protection contre la suppression.
3. Dans la page principale Clusters, choisissez à nouveau votre cluster Amazon DocumentDB.
4. Sélectionnez Actions, Supprimer.
5. Pour Créer un instantané final du cluster, sélectionnez Non.
6. Saisissez **`delete`** dans le champ de saisie de texte.
7. Sélectionnez Delete (Supprimer).

Pour supprimer le secret dans Secrets Manager

1. Ouvrez la console Secrets Manager.
2. Sélectionnez le secret que vous avez créé pour ce tutoriel.
3. Choisissez Actions, Supprimer le secret.
4. Choisissez Schedule deletion (Planifier la suppression).

Utilisation d'AWS Lambda avec Amazon DynamoDB

> **Note**
>
> Si vous voulez envoyer des données à une cible autre qu'une fonction Lambda ou enrichir les données avant de les envoyer, consultez Amazon EventBridge Pipes.

Vous pouvez utiliser une fonction AWS Lambda pour traiter des enregistrements dans un flux Amazon DynamoDB. DynamoDB Streams vous permet de déclencher une fonction Lambda pour effectuer un travail supplémentaire chaque fois qu'une table DynamoDB est mise à jour.

Lorsque vous traitez des flux DynamoDB, vous devez implémenter une logique de réponse partielle de lot afin d'éviter que les enregistrements traités avec succès soient de nouveau essayés en cas d'échec de certains enregistrements d'un lot. L'utilitaire de traitement par lots de Powertools pour AWS Lambda est disponible en Python, TypeScript, .NET et Java et simplifie cette implémentation en gérant automatiquement la logique de réponse partielle de lot, réduisant ainsi le temps de développement et améliorant la fiabilité.

Rubriques

- Flux d'interrogation et de mise en lots
- Positions de départ des interrogations et des flux
- Lecteurs simultanés d'une partition dans DynamoDB Streams
- Exemple d'évènement
- Traiter les enregistrements DynamoDB avec Lambda
- Configuration d'une réponse par lots partielle avec DynamoDB et Lambda
- Retenir les enregistrements ignorés pour une source d'événement DynamoDB dans Lambda
- Implémentation du traitement des flux DynamoDB avec état dans Lambda
- Paramètres Lambda pour les mappages des sources d'événement Amazon DynamoDB
- Utilisation du filtrage des événements avec une source d'événement DynamoDB
- Tutoriel : Utilisation AWS Lambda avec les flux Amazon DynamoDB

Flux d'interrogation et de mise en lots

Lambda interroge les partitions de votre flux DynamoDB en quête d'enregistrements à une fréquence de base de quatre fois par seconde. Lorsque des enregistrements sont disponibles, Lambda invoque votre fonction et attend le résultat. Si le traitement réussit, Lambda reprend l'interrogation jusqu'à recevoir plus d'enregistrements.

Par défaut, Lambda invoque votre fonction dès que des enregistrements sont disponibles. Si le lot que Lambda lit à partir de la source d'événements ne comprend qu'un seul enregistrement, Lambda envoie un seul registre à la fonction. Pour éviter d'invoquer la fonction avec un petit nombre de registres, vous pouvez indiquer à la source d'événement de les mettre en mémoire tampon pendant 5 minutes en configurant une fenêtre de traitement par lots. Avant d'invoquer la fonction, Lambda continue de lire les registres de la source d'événements jusqu'à ce qu'il ait rassemblé un lot complet, que la fenêtre de traitement par lot expire ou que le lot atteigne la limite de charge utile de 6 Mo. Pour de plus amples informations, consultez Comportement de traitement par lots.

> **Warning**
>
> Les mappages des sources d'événements Lambda traitent chaque événement au moins une fois, et le traitement des enregistrements peut être dupliqué. Pour éviter les problèmes potentiels liés à des événements dupliqués, nous vous recommandons vivement de rendre votre code de fonction idempotent. Pour en savoir plus, consultez Comment puis-je rendre ma fonction Lambda idempotente ? dans le Centre de connaissances AWS.

Lambda envoi le prochain lot pour traitement sans attendre que les extensions configurées soient terminées. En d'autres termes, vos extensions peuvent continuer à s'exécuter pendant que Lambda traite le prochain lot d'enregistrements. Cela peut causer des problèmes de limitation si vous enfreignez l'un des paramètres ou l'une des limites de simultanéité de votre compte. Pour détecter s'il s'agit d'un problème éventuel, surveillez vos fonctions et vérifiez si vous observez des métriques de simultanéité plus élevées que prévu pour votre mappage des sources d'événements. En raison de la brièveté des intervalles entre les invocations, Lambda peut brièvement signaler une utilisation simultanée supérieure au nombre de partitions. Cela peut être vrai même pour les fonctions Lambda sans extensions.

Configurez le paramètre ParallelizationFactor pour traiter une partition d'un flux DynamoDB avec plusieurs invocations Lambda simultanément. Vous pouvez spécifier le nombre de lots simultanés que Lambda interroge à partir d'une partition via un facteur de parallélisation de 1 (par défaut) à 10.

Par exemple, quand vous définissez `ParallelizationFactor` sur 2, vous pouvez avoir jusqu'à 200 invocations Lambda simultanés pour traiter 100 partitions de données DynamoDB (bien que, dans la réalité, la métrique `ConcurrentExecutions` puisse indiquer une valeur différente). Cela permet d'augmenter le débit de traitement quand le volume de données est volatil et que la valeur du paramètre IteratorAge est élevée. Lorsque vous augmentez le nombre de lots simultanés par partition, Lambda assure toujours un traitement dans l'ordre au niveau de l'élément (partition et clé de tri).

Positions de départ des interrogations et des flux

Sachez que l'interrogation des flux lors des mises à jour et de la création du mappage des sources d'événements est finalement cohérente.

- Lors de la création du mappage des sources d'événements, le démarrage de l'interrogation des événements depuis le flux peut prendre plusieurs minutes.
- Lors des mises à jour du mappage des sources d'événements, l'arrêt et le redémarrage de l'interrogation des événements depuis le flux peuvent prendre plusieurs minutes.

Ce comportement signifie que si vous spécifiez `LATEST` comme position de départ du flux, le mappage des sources d'événements peut manquer des événements lors de la création ou des mises à jour. Pour vous assurer de ne manquer aucun événement, spécifiez la position de départ du flux comme `TRIM_HORIZON`.

Lecteurs simultanés d'une partition dans DynamoDB Streams

Pour les tables à région unique qui ne sont pas des tables globales, vous pouvez concevoir jusqu'à deux fonctions Lambda pour lire la même partition DynamoDB Streams simultanément. Le dépassement de cette limite peut se traduire par une limitation de la demande. Pour les tables globales, nous vous recommandons de limiter le nombre de fonctions simultanées à une seule pour éviter la limitation des demandes.

Exemple d'évènement

Example

```
{
  "Records": [
    {
      "eventID": "1",
```

```
      "eventVersion": "1.0",
      "dynamodb": {
        "Keys": {
          "Id": {
            "N": "101"
          }
        },
        "NewImage": {
          "Message": {
            "S": "New item!"
          },
          "Id": {
            "N": "101"
          }
        },
        "StreamViewType": "NEW_AND_OLD_IMAGES",
        "SequenceNumber": "111",
        "SizeBytes": 26
      },
      "awsRegion": "us-west-2",
      "eventName": "INSERT",
      "eventSourceARN": "arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525",
      "eventSource": "aws:dynamodb"
    },
    {
      "eventID": "2",
      "eventVersion": "1.0",
      "dynamodb": {
        "OldImage": {
          "Message": {
            "S": "New item!"
          },
          "Id": {
            "N": "101"
          }
        },
        "SequenceNumber": "222",
        "Keys": {
          "Id": {
            "N": "101"
          }
        },
        "SizeBytes": 59,
```

```
        "NewImage": {
          "Message": {
            "S": "This item has changed"
          },
          "Id": {
            "N": "101"
          }
        },
        "StreamViewType": "NEW_AND_OLD_IMAGES"
      },
      "awsRegion": "us-west-2",
      "eventName": "MODIFY",
      "eventSourceARN": "arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525",
      "eventSource": "aws:dynamodb"
    }
  ]}
```

Traiter les enregistrements DynamoDB avec Lambda

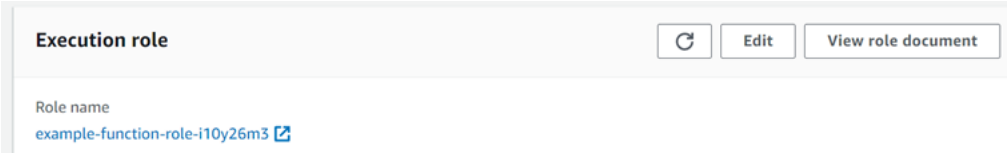
Créez un mappage de source d'événement pour indiquer à Lambda d'envoyer des enregistrements de votre flux à une fonction Lambda. Vous pouvez créer plusieurs mappages de source d'événement pour traiter les mêmes données avec plusieurs fonctions Lambda, ou traiter des éléments de plusieurs flux avec une seule fonction.

Vous pouvez configurer des mappages de sources d'événements pour traiter les enregistrements d'un flux dans un autre Compte AWS. Pour en savoir plus, veuillez consulter la section the section called "Mappages entre comptes".

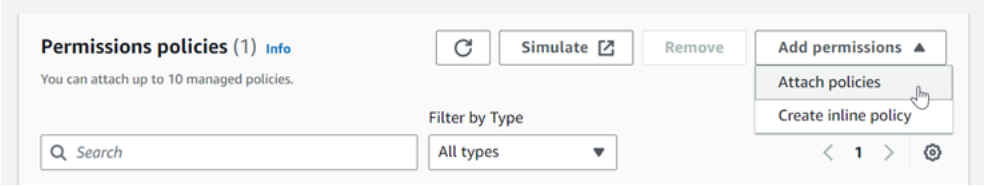
Pour configurer votre fonction de manière à lire depuis DynamoDB Streams, associez la politique AWS gérée AWSLambdaDynamo Role à votre DBExecution rôle d'exécution, puis créez un déclencheur DynamoDB.

Pour ajouter des autorisations et créer un déclencheur

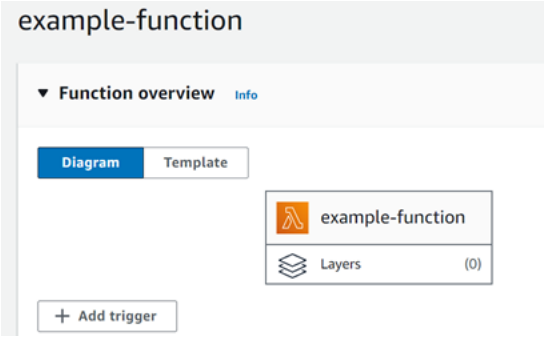
1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom d'une fonction.
3. Choisissez l'onglet Configuration, puis Permissions (Autorisations).
4. Sous Nom du rôle, cliquez sur le lien vers votre rôle d'exécution. Ce lien ouvre le rôle dans la console IAM.



5. Choisissez Ajouter des autorisations, puis Attacher des politiques.



6. Dans le champ de recherche, entrez `AWSLambdaDynamoDBExecutionRole`. Ajoutez cette politique à votre rôle d'exécution. Il s'agit d'une politique AWS gérée qui contient les autorisations dont votre fonction a besoin pour lire le flux DynamoDB. Pour plus d'informations sur cette politique, consultez AWSLambdaDynamo DBExecution Role dans le manuel AWS Managed Policy Reference.
7. Revenez à votre fonction dans la console Lambda. Sous Function overview (Vue d'ensemble de la fonction), choisissez Add trigger (Ajouter un déclencheur).



8. Choisissez un type de déclencheur.
9. Configurez les options requises, puis choisissez Add (Ajouter).

Lambda prend en charge les options suivantes pour les sources d'événement DynamoDB :

Options de source d'événement

- DynamoDB table (Table DynamoDB) – Table DynamoDB à partir de laquelle lire les enregistrements.
- Batch size (Taille de lot) – Nombre d'enregistrements par lot à envoyer à la fonction, jusqu'à 10 000. Lambda transmet tous les enregistrements du lot à la fonction en une seule invocation, tant

que la taille totale des événements ne dépasse pas la limite de charge utile pour une invocation synchrone (6 Mo).

- Batch window (Fenêtre de traitement par lots) – Intervalle de temps maximum (en secondes) pour collecter des enregistrements avant d'invoquer la fonction.
- Starting position (Position de départ) – Traiter uniquement les nouveaux enregistrements, ou tous enregistrement existants.
  - Latest (Derniers) – Traiter les nouveaux enregistrements ajoutés au flux.
  - Trim horizon (Supprimer l'horizon) – Traiter tous les enregistrements figurant dans le flux.

  Après le traitement de tous les enregistrements existants, la fonction est à jour et continue à traiter les nouveaux enregistrements.
- Destination en cas d'échec : une file d'attente SQS standard ou rubrique SNS standard pour les enregistrements qui ne peuvent pas être traités. Quand Lambda écarte un lot d'enregistrements qui est trop ancien ou qui a épuisé toutes les tentatives, il envoie les détails du lot à la file d'attente ou à la rubrique.
- Retry attempts (Nouvelles tentatives) – Nombre maximum de nouvelles tentatives que Lambda effectue quand la fonction renvoie une erreur. Cela ne s'applique pas aux erreurs ou limitations de service où le lot n'a pas atteint la fonction.
- Maximum age of record (Âge maximum d'enregistrement) – Âge maximum d'un enregistrement que Lambda envoie à votre fonction.
- Split batch on error (Fractionner le lot en cas d'erreur) – Quand la fonction renvoie une erreur, diviser le lot en deux avant d'effectuer une nouvelle tentative. Le paramètre de taille de lot d'origine reste inchangé.
- Concurrent batches per shard (Lots simultanés par partition) – Traite simultanément plusieurs lots de la même partition.
- Enabled (Activé) – Valeur définie sur VRAI pour activer le mappage de source d'événement. Définissez la valeur sur « false » pour arrêter le traitement des enregistrements. Lambda garde une trace du dernier enregistrement traité et reprend le traitement à ce point lorsque le mappage est réactivé.

> **Note**
>
> Les appels d' GetRecords API invoqués par Lambda dans le cadre des déclencheurs DynamoDB ne vous sont pas facturés.

Pour gérer ultérieurement la configuration de la source d'événement, choisissez le déclencheur dans le concepteur.

Création d'un mappage des sources d'événements entre comptes

Amazon DynamoDB prend désormais en charge les politiques basées sur les ressources. Grâce à cette fonctionnalité, vous pouvez traiter les données d'un flux DynamoDB dans un compte Compte AWS avec une fonction Lambda dans un autre compte.

Pour créer un mappage de source d'événements pour votre fonction Lambda à l'aide d'un flux DynamoDB dans un autre Compte AWS, vous devez configurer le flux à l'aide d'une politique basée sur les ressources afin d'autoriser votre fonction Lambda à lire des enregistrements. Pour savoir comment configurer votre flux pour un accès entre comptes, consultez Partager l'accès avec les fonctions Lambda entre comptes dans le guide du développeur Amazon DynamoDB.

Une fois que vous avez configuré votre flux avec une politique basée sur les ressources qui donne à votre fonction Lambda les autorisations requises, créez le mappage des sources d'événements avec l'ARN de votre flux entre comptes. Vous pouvez trouver l'ARN du flux sous l'onglet Exportations et flux du tableau dans la console DynamoDB multicomptes.

Lorsque vous utilisez la console Lambda, collez l'ARN du flux directement dans le champ de saisie de la table DynamoDB sur la page de création du mappage des sources d'événements.

Remarque : Les déclencheurs interrégionaux ne sont pas pris en charge.

Configuration d'une réponse par lots partielle avec DynamoDB et Lambda

Lors de l'utilisation et du traitement de données de streaming à partir d'une source d'événement, par défaut, Lambda n'effectue un point de contrôle sur le numéro de séquence le plus élevé d'un lot que lorsque celui-ci est un succès complet. Lambda traite tous les autres résultats comme un échec complet et recommence à traiter le lot jusqu'à atteindre la limite de nouvelles tentatives. Pour autoriser des succès partiels lors du traitement des lots à partir d'un flux, activez `ReportBatchItemFailures`. Autoriser des succès partiels peut permettre de réduire le nombre de nouvelles tentatives sur un enregistrement, mais cela n'empêche pas entièrement la possibilité de nouvelles tentatives dans un enregistrement réussi.

Pour activer `ReportBatchItemFailures`, incluez la valeur enum **`ReportBatchItemFailures`** dans la listeFunctionResponseTypes. Cette liste indique quels types de réponse sont activés pour votre fonction. Vous pouvez configurer cette liste lorsque vous créez ou mettez à jour un mappage des sources d'événements.

> **Note**
>
> Même lorsque le code de votre fonction renvoie des réponses d'échec partiel de lot, ces réponses ne seront pas traitées par Lambda à moins que la fonctionnalité `ReportBatchItemFailures` soit explicitement activée pour votre mappage des sources d'événements.

Syntaxe du rapport

Lors de la configuration des rapports d'échec d'articles de lot, la classe `StreamsEventResponse` est renvoyée avec une liste d'échecs d'articles de lot. Vous pouvez utiliser un objet `StreamsEventResponse` pour renvoyer le numéro de séquence du premier enregistrement ayant échoué dans le lot. Vous pouvez également créer votre classe personnalisée en utilisant la syntaxe de réponse correcte. La structure JSON suivante montre la syntaxe de réponse requise :

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "<SequenceNumber>"
        }
    ]
}
```

> **Note**
>
> Si le tableau `batchItemFailures` contient plusieurs éléments, Lambda utilise l'enregistrement portant le numéro de séquence le plus bas comme point de contrôle. Lambda réessaie ensuite tous les enregistrements à partir de ce point de contrôle.

Conditions de réussite et d'échec

Lambda traite un lot comme un succès complet si vous renvoyez l'un des éléments suivants :

- Une liste `batchItemFailure` vide
- Une liste `batchItemFailure` nulle
- Une `EventResponse` vide

- Un nu `EventResponse`

Lambda traite un lot comme un échec complet si vous renvoyez l'un des éléments suivants :

- Une chaîne vid `itemIdentifier`
- Un `itemIdentifier` nul
- Un `itemIdentifier` avec un nom de clé incorrect

Lambda effectue des nouvelles tentatives en cas d'échec en fonction de votre stratégie de nouvelle tentative.

Diviser un lot

Si votre invocation échoue et que `BisectBatchOnFunctionError` est activé, le lot est divisé en deux quel que soit votre paramètre `ReportBatchItemFailures`.

Quand une réponse de succès partiel de lot est reçue et que les paramètres `BisectBatchOnFunctionError` et `ReportBatchItemFailures` sont activés, le lot est divisé au numéro de séquence renvoyé, et Lambda n'effectue de nouvelle tentative que sur les enregistrements restants.

Pour simplifier la mise en œuvre de la logique de réponse partielle par lots, pensez à utiliser l'utilitaire Batch Processor de Powertools for AWS Lambda, qui gère automatiquement ces complexités pour vous.

Voici quelques exemples de code de fonction qui renvoie la liste des messages ayant échoué IDs dans le lot :

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots DynamoDB avec Lambda à l'aide de .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public StreamsEventResponse FunctionHandler(DynamoDBEvent dynamoEvent,
 ILambdaContext context)

    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");
        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>();
        StreamsEventResponse streamsEventResponse = new StreamsEventResponse();

        foreach (var record in dynamoEvent.Records)
        {
            try
            {
                var sequenceNumber = record.Dynamodb.SequenceNumber;
                context.Logger.LogInformation(sequenceNumber);
            }
            catch (Exception ex)
            {
                context.Logger.LogError(ex.Message);
                batchItemFailures.Add(new StreamsEventResponse.BatchItemFailure()
 { ItemIdentifier = record.Dynamodb.SequenceNumber });
            }
        }

        if (batchItemFailures.Count > 0)
        {
```

```
            streamsEventResponse.BatchItemFailures = batchItemFailures;
        }

        context.Logger.LogInformation("Stream processing complete.");
        return streamsEventResponse;
    }
}
```

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots DynamoDB avec Lambda à l'aide de Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

type BatchItemFailure struct {
 ItemIdentifier string `json:"ItemIdentifier"`
}

type BatchResult struct {
 BatchItemFailures []BatchItemFailure `json:"BatchItemFailures"`
}

func HandleRequest(ctx context.Context, event events.DynamoDBEvent)
 (*BatchResult, error) {
 var batchItemFailures []BatchItemFailure
 curRecordSequenceNumber := ""
```

```
 for _, record := range event.Records {
  // Process your record
  curRecordSequenceNumber = record.Change.SequenceNumber
 }

 if curRecordSequenceNumber != "" {
  batchItemFailures = append(batchItemFailures, BatchItemFailure{ItemIdentifier:
 curRecordSequenceNumber})
 }

 batchResult := BatchResult{
  BatchItemFailures: batchItemFailures,
 }

 return &batchResult, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots DynamoDB avec Lambda à l'aide de Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import com.amazonaws.services.lambda.runtime.events.models.dynamodb.StreamRecord;
```

```
import java.util.ArrayList;
import java.util.List;

public class ProcessDynamodbRecords implements RequestHandler<DynamodbEvent,
 StreamsEventResponse> {

    @Override
    public StreamsEventResponse handleRequest(DynamodbEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (DynamodbEvent.DynamodbStreamRecord dynamodbStreamRecord :
 input.getRecords()) {
          try {
                //Process your record
                StreamRecord dynamodbRecord = dynamodbStreamRecord.getDynamodb();
                curRecordSequenceNumber = dynamodbRecord.getSequenceNumber();

            } catch (Exception e) {
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse();
    }
}
```

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signaler les défaillances d'éléments de lot DynamoDB avec Lambda à l'aide de. JavaScript

```
export const handler = async (event) => {
  const records = event.Records;
  let curRecordSequenceNumber = "";

  for (const record of records) {
    try {
      // Process your record
      curRecordSequenceNumber = record.dynamodb.SequenceNumber;
    } catch (e) {
      // Return failed record's sequence number
      return { batchItemFailures: [{ itemIdentifier:
 curRecordSequenceNumber }] };
    }
  }

  return { batchItemFailures: [] };
};
```

Signaler les défaillances d'éléments de lot DynamoDB avec Lambda à l'aide de. TypeScript

```
import {
  DynamoDBBatchResponse,
  DynamoDBBatchItemFailure,
  DynamoDBStreamEvent,
} from "aws-lambda";

export const handler = async (
  event: DynamoDBStreamEvent
): Promise<DynamoDBBatchResponse> => {
```

```
  const batchItemFailures: DynamoDBBatchItemFailure[] = [];
  let curRecordSequenceNumber;

  for (const record of event.Records) {
    curRecordSequenceNumber = record.dynamodb?.SequenceNumber;

    if (curRecordSequenceNumber) {
      batchItemFailures.push({
        itemIdentifier: curRecordSequenceNumber,
      });
    }
  }

  return { batchItemFailures: batchItemFailures };
};
```

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots DynamoDB avec Lambda à l'aide de PHP.

```
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
```

```
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $dynamoDbEvent = new DynamoDbEvent($event);
        $this->logger->info("Processing records");

        $records = $dynamoDbEvent->getRecords();
        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
```

```
return new Handler($logger);
```

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots DynamoDB avec Lambda à l'aide de Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["dynamodb"]["SequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots DynamoDB avec Lambda à l'aide de Ruby.

```
def lambda_handler(event:, context:)
    records = event["Records"]
    cur_record_sequence_number = ""

    records.each do |record|
      begin
        # Process your record
        cur_record_sequence_number = record["dynamodb"]["SequenceNumber"]
      rescue StandardError => e
        # Return failed record's sequence number
        return {"batchItemFailures" => [{"itemIdentifier" =>
 cur_record_sequence_number}]}
      end
    end

    {"batchItemFailures" => []}
  end
```

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots DynamoDB avec Lambda à l'aide de Rust.

```
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord, StreamRecord},
    streams::{DynamoDbBatchItemFailure, DynamoDbEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

/// Process the stream record
fn process_record(record: &EventRecord) -> Result<(), Error> {
    let stream_record: &StreamRecord = &record.change;

    // process your stream record here...
    tracing::info!("Data: {:?}", stream_record);

    Ok(())
}

/// Main Lambda handler here...
async fn function_handler(event: LambdaEvent<Event>) ->
 Result<DynamoDbEventResponse, Error> {
    let mut response = DynamoDbEventResponse {
        batch_item_failures: vec![],
    };

    let records = &event.payload.records;

    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in records {
        tracing::info!("EventId: {}", record.event_id);

        // Couldn't find a sequence number
        if record.change.sequence_number.is_none() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: Some("".to_string()),
            });
            return Ok(response);
        }

        // Process your record here...
```

```
        if process_record(record).is_err() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: record.change.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!("Successfully processed {} record(s)", records.len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

Utilisation de Powertools pour le AWS Lambda traitement par lots

L'utilitaire de traitement par lots de Powertools for gère AWS Lambda automatiquement la logique de réponse partielle par lots, réduisant ainsi la complexité de la mise en œuvre du signalement des défaillances par lots. Voici des exemples d'utilisation de l'utilitaire de traitement par lots :

Python

> **Note**
>
> Pour des exemples complets et des instructions de configuration, consultez la documentation de l'utilitaire de traitement par lots.

Traitement des enregistrements AWS Lambda de flux DynamoDB à l'aide d'un processeur par lots.

```
import json
from aws_lambda_powertools import Logger
from aws_lambda_powertools.utilities.batch import BatchProcessor, EventType,
 process_partial_response
from aws_lambda_powertools.utilities.data_classes import DynamoDBStreamEvent
from aws_lambda_powertools.utilities.typing import LambdaContext

processor = BatchProcessor(event_type=EventType.DynamoDBStreams)
logger = Logger()

def record_handler(record):
    logger.info(record)
    # Your business logic here
    # Raise an exception to mark this record as failed

def lambda_handler(event, context: LambdaContext):
    return process_partial_response(
        event=event,
        record_handler=record_handler,
        processor=processor,
        context=context
    )
```

TypeScript

> **Note**
>
> Pour des exemples complets et des instructions de configuration, consultez la documentation de l'utilitaire de traitement par lots.

Traitement des enregistrements AWS Lambda de flux DynamoDB à l'aide d'un processeur par lots.

```
import { BatchProcessor, EventType, processPartialResponse } from '@aws-lambda-
powertools/batch';
import { Logger } from '@aws-lambda-powertools/logger';
import type { DynamoDBStreamEvent, Context } from 'aws-lambda';

const processor = new BatchProcessor(EventType.DynamoDBStreams);
const logger = new Logger();

const recordHandler = async (record: any): Promise<void> => {
    logger.info('Processing record', { record });
    // Your business logic here
    // Throw an error to mark this record as failed
};

export const handler = async (event: DynamoDBStreamEvent, context: Context) => {
    return processPartialResponse(event, recordHandler, processor, {
        context,
    });
};
```

Java

> **Note**
>
> Pour des exemples complets et des instructions de configuration, consultez la documentation de l'utilitaire de traitement par lots.

Traitement des enregistrements AWS Lambda de flux DynamoDB à l'aide d'un processeur par lots.

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import software.amazon.lambda.powertools.batch.BatchMessageHandlerBuilder;
import software.amazon.lambda.powertools.batch.handler.BatchMessageHandler;
```

```
public class DynamoDBStreamBatchHandler implements RequestHandler<DynamodbEvent,
 StreamsEventResponse> {

    private final BatchMessageHandler<DynamodbEvent, StreamsEventResponse> handler;

    public DynamoDBStreamBatchHandler() {
        handler = new BatchMessageHandlerBuilder()
                .withDynamoDbBatchHandler()
                .buildWithRawMessageHandler(this::processMessage);
    }

    @Override
    public StreamsEventResponse handleRequest(DynamodbEvent ddbEvent, Context
 context) {
        return handler.processBatch(ddbEvent, context);
    }

    private void processMessage(DynamodbEvent.DynamodbStreamRecord
 dynamodbStreamRecord, Context context) {
        // Process the change record
    }
}
```

.NET

> **Note**
>
> Pour des exemples complets et des instructions de configuration, consultez la documentation de l'utilitaire de traitement par lots.

Traitement des enregistrements AWS Lambda de flux DynamoDB à l'aide d'un processeur par lots.

```
using System;
using System.Threading;
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;
using Amazon.Lambda.Serialization.SystemTextJson;
using AWS.Lambda.Powertools.BatchProcessing;
```

```
[assembly: LambdaSerializer(typeof(DefaultLambdaJsonSerializer))]

namespace HelloWorld;

public class Customer
{
    public string? CustomerId { get; set; }
    public string? Name { get; set; }
    public string? Email { get; set; }
    public DateTime CreatedAt { get; set; }
}

internal class TypedDynamoDbRecordHandler : ITypedRecordHandler<Customer>
{
    public async Task<RecordHandlerResult> HandleAsync(Customer customer,
 CancellationToken cancellationToken)
    {
        if (string.IsNullOrEmpty(customer.Email))
        {
            throw new ArgumentException("Customer email is required");
        }

        return await Task.FromResult(RecordHandlerResult.None);
    }
}

public class Function
{
    [BatchProcessor(TypedRecordHandler = typeof(TypedDynamoDbRecordHandler))]
    public BatchItemFailuresResponse HandlerUsingTypedAttribute(DynamoDBEvent _)
    {
        return TypedDynamoDbStreamBatchProcessor.Result.BatchItemFailuresResponse;
    }
}
```

Retenir les enregistrements ignorés pour une source d’événement DynamoDB dans Lambda

La gestion des erreurs pour les mappages des sources d’événements DynamoDB n’est pas la même selon si l’erreur se produit avant que la fonction ne soit invoquée ou pendant l’invocation de la fonction :

- Avant l'appel : si un mappage de source d'événement Lambda ne parvient pas à appeler la fonction en raison d'un ralentissement ou d'autres problèmes, il réessaie jusqu'à ce que les enregistrements expirent ou dépassent l'âge maximum configuré sur le mappage de source d'événement (). MaximumRecordAgeInSeconds
- Pendant l'appel : si la fonction est invoquée mais renvoie une erreur, Lambda réessaie jusqu'à ce que les enregistrements expirent, dépassent l'âge maximum MaximumRecordAgeInSeconds() ou atteignent le quota de nouvelles tentatives configuré (). MaximumRetryAttempts Pour les erreurs de fonctionnement, vous pouvez également configurer BisectBatchOnFunctionError, ce qui divise un lot défaillant en deux lots plus petits, isolant ainsi les mauvais enregistrements et évitant les délais d'attente. Le fractionnement des lots ne consomme pas le quota de nouvelles tentatives.

Si les mesures de gestion des erreurs échouent, Lambda ignore les enregistrements et poursuit le traitement des lots du flux. Avec les paramètres par défaut, cela signifie qu'un enregistrement défectueux peut bloquer le traitement sur la partition affectée pendant jusqu'à une journée. Pour éviter cela, configurez le mappage de source d'événement de votre fonction avec un nombre raisonnable de nouvelles tentatives et un âge maximum d'enregistrement correspondant à votre cas d'utilisation.

Configuration des destinations pour les invocations ayant échoué

Pour retenir les enregistrements des invocations de mappage de sources d'événements qui ont échoué, ajoutez une destination au mappage des sources d'événements de votre fonction. Chaque enregistrement envoyé à la destination est un document JSON contenant les métadonnées sur l'invocation ayant échoué. Pour les destinations Amazon S3, Lambda envoie également l'intégralité de l'enregistrement d'invocation avec les métadonnées. Vous pouvez configurer n'importe quelle rubrique Amazon SNS, n'importe quelle file d'attente Amazon SQS, n'importe quel compartiment Amazon S3 ou Kafka comme destination.

Avec les destinations Amazon S3, vous pouvez utiliser la fonctionnalité Notifications d'événements Amazon S3 pour recevoir des notifications lorsque des objets sont chargés dans votre compartiment S3 de destination. Vous pouvez également configurer les notifications d'événements S3 pour invoquer une autre fonction Lambda afin d'effectuer un traitement automatique des lots ayant échoué.

Votre rôle d'exécution doit disposer d'autorisations pour la destination :

- Pour une destination SQS : sqs : SendMessage
- Pour une destination SNS : SNS:Publish
- Pour une destination S3 : s3 : PutObject et s3 : ListBucket

- Pour une destination Kafka : kafka-cluster : WriteData

Vous pouvez configurer un sujet Kafka comme destination en cas d'échec pour vos mappages de sources d'événements Kafka. Lorsque Lambda ne parvient pas à traiter les enregistrements après avoir épuisé toutes les tentatives ou lorsque les enregistrements dépassent l'âge maximum, Lambda envoie les enregistrements ayant échoué à la rubrique Kafka spécifiée pour un traitement ultérieur. Reportez-vous à the section called “Destination de Kafka en cas d'échec”.

Si vous avez activé le chiffrement avec votre propre clé KMS pour une destination S3, le rôle d'exécution de votre fonction doit également être autorisé à appeler kms : GenerateDataKey. Si la clé KMS et la destination du compartiment S3 se trouvent dans un compte différent de celui de votre fonction Lambda et de votre rôle d'exécution, configurez la clé KMS pour qu'elle approuve le rôle d'exécution à autoriser. kms: GenerateDataKey

Pour configurer une destination en cas de panne à l’aide de la console, procédez comme suit :

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Sous Function overview (Vue d’ensemble de la fonction), choisissez Add destination (Ajouter une destination).
4. Pour Source, choisissez Invocation du mappage des sources d’événements.
5. Pour le mappage des sources d’événements, choisissez une source d’événements configurée pour cette fonction.
6. Pour Condition, sélectionnez En cas d’échec. Pour les invocations de mappage des sources d’événements, il s’agit de la seule condition acceptée.
7. Pour Type de destination, choisissez le type de destination auquel Lambda envoie les enregistrements d’invocation.
8. Pour Destination, choisissez une ressource.
9. Choisissez Enregistrer.

Vous pouvez également configurer une destination en cas de panne à l'aide de AWS Command Line Interface (AWS CLI). Par exemple, la create-event-source-mappingcommande suivante ajoute un mappage de source d'événement avec une destination SQS en cas de défaillance pour : `MyFunction`

```
aws lambda create-event-source-mapping \
```

```
--function-name "MyFunction" \
--event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-
east-1:123456789012:dest-queue"}}'
```

La update-event-source-mappingcommande suivante met à jour le mappage d'une source d'événements afin d'envoyer les enregistrements d'invocation ayant échoué vers une destination SNS après deux tentatives, ou si les enregistrements datent de plus d'une heure.

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--maximum-retry-attempts 2 \
--maximum-record-age-in-seconds 3600 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sns:us-
east-1:123456789012:dest-topic"}}'
```

Les paramètres mis à jour sont appliqués de façon asynchrone et ne sont pas reflétés dans la sortie tant que le processus n'est pas terminé. Utilisez la commande get-event-source-mapping pour afficher l'état actuel.

Pour supprimer une destination, entrez une chaîne vide comme argument du paramètre `destination-config` :

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

Pratiques exemplaires en matière de sécurité pour les destinations Amazon S3

La suppression d'un compartiment S3 configuré comme destination sans supprimer la destination de la configuration de votre fonction peut engendrer un risque de sécurité. Si un autre utilisateur connaît le nom de votre compartiment de destination, il peut recréer le compartiment dans son Compte AWS. Les enregistrements des invocations ayant échoué seront envoyés dans son compartiment, exposant potentiellement les données de votre fonction.

> **Warning**
>
> Pour vous assurer que les enregistrements d'invocation de votre fonction ne peuvent pas être envoyés vers un compartiment S3 d'un autre Compte AWS, ajoutez une condition au rôle

d'exécution de votre fonction qui limite `s3:PutObject` les autorisations aux compartiments de votre compte.

L'exemple suivant présente une politique IAM qui limite les autorisations `s3:PutObject` de votre fonction aux seuls compartiments de votre compte. Cette politique donne également à Lambda l'autorisation `s3:ListBucket` dont il a besoin pour utiliser un compartiment S3 comme destination.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "S3BucketResourceAccountWrite",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::*/*",
                "arn:aws:s3:::*"
            ],
            "Condition": {
                "StringEquals": {
                    "s3:ResourceAccount": "111122223333"
                }
            }
        }
    ]
}
```

Pour ajouter une politique d'autorisations au rôle d'exécution de votre fonction à l'aide du AWS Management Console ou AWS CLI, reportez-vous aux instructions des procédures suivantes :

Console

Pour ajouter une politique d'autorisations au rôle d'exécution d'une fonction (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction Lambda dont vous voulez modifier le rôle d'exécution.
3. Sous l'onglet Configuration, sélectionnez Autorisations.

4. Sous l'onglet Rôle d'exécution, sélectionnez le Nom du rôle de votre fonction pour ouvrir la page de console IAM du rôle.
5. Ajoutez une politique d'autorisations de au rôle en procédant comme suit :

    a. Dans le volet Politiques d'autorisations, choisissez Ajouter des autorisations, puis Créer une politique en ligne.

    b. Dans l'Éditeur de politique, sélectionnez JSON.

    c. Collez la politique que vous souhaitez ajouter dans l'éditeur (en remplacement du JSON existant), puis choisissez Suivant.

    d. Sous Détails de la politique, saisissez un Nom de la politique.

    e. Choisissez Create Policy (Créer une politique).

AWS CLI

Pour ajouter une politique d'autorisations au rôle d'exécution d'une fonction (CLI)

1. Créez un document de politique JSON avec les autorisations requises et enregistrez-le dans un répertoire local.
2. Utilisez la commande `put-role-policy` de la CLI IAM pour ajouter des autorisations au rôle d'exécution de votre fonction. Exécutez la commande suivante depuis le répertoire dans lequel vous avez enregistré votre document de politique JSON et remplacez le nom du rôle, le nom de la politique et le document de politique par vos propres valeurs.

```
aws iam put-role-policy \
--role-name my_lambda_role \
--policy-name LambdaS3DestinationPolicy \
--policy-document file://my_policy.json
```

Exemple d'enregistrement d'invocation Amazon SNS et Amazon SQS

L'exemple suivant illustre un enregistrement d'invocation que Lambda envoie à une destination SQS ou SNS pour un flux DynamoDB.

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:myfunction",
```

```
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:13:49.717Z",
    "DDBStreamBatchInfo": {
        "shardId": "shardId-00000001573689847184-864758bb",
        "startSequenceNumber": "800000000003126276362",
        "endSequenceNumber": "800000000003126276362",
        "approximateArrivalOfFirstRecord": "2019-11-14T00:13:19Z",
        "approximateArrivalOfLastRecord": "2019-11-14T00:13:19Z",
        "batchSize": 1,
        "streamArn": "arn:aws:dynamodb:us-east-2:123456789012:table/mytable/
stream/2019-11-14T00:04:06.388"
    }
}
```

Vous pouvez utiliser ces informations pour récupérer les enregistrements concernés à partir du flux à des fins de résolution de problèmes. Les enregistrements réels n'étant pas inclus, vous devez les récupérer du flux avant qu'ils expirent et soient perdus.

Exemple d'enregistrement d'invocation Amazon S3

L'exemple suivant illustre un enregistrement d'invocation que Lambda envoie à un compartiment S3 pour un flux DynamoDB. Outre tous les champs de l'exemple précédent pour les destinations SQS et SNS, le champ `payload` contient l'enregistrement d'invocation d'origine sous forme de chaîne JSON échappée.

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
```

```
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:13:49.717Z",
    "DDBStreamBatchInfo": {
        "shardId": "shardId-00000001573689847184-864758bb",
        "startSequenceNumber": "80000000003126276362",
        "endSequenceNumber": "80000000003126276362",
        "approximateArrivalOfFirstRecord": "2019-11-14T00:13:19Z",
        "approximateArrivalOfLastRecord": "2019-11-14T00:13:19Z",
        "batchSize": 1,
        "streamArn": "arn:aws:dynamodb:us-east-2:123456789012:table/mytable/
stream/2019-11-14T00:04:06.388"
    },
    "payload": "<Whole Event>" // Only available in S3
}
```

L'objet S3 contenant l'enregistrement d'invocation utilise la convention de dénomination suivante :

```
aws/lambda/<ESM-UUID>/<shardID>/YYYY/MM/DD/YYYY-MM-DDTHH.MM.SS-<Random UUID>
```

Implémentation du traitement des flux DynamoDB avec état dans Lambda

Les fonctions Lambda peuvent exécuter des applications de traitement de flux continu. Un flux représente des données illimitées qui circulent en continu dans votre application. Pour analyser les informations provenant de cette entrée de mise à jour continue, vous pouvez lier les enregistrements inclus à l'aide d'une fenêtre de temps définie.

Les fenêtres bascules sont des fenêtres temporelles distinctes qui s'ouvrent et se ferment à intervalles réguliers. Par défaut, les invocations Lambda sont sans état : vous ne pouvez pas les utiliser pour traiter des données sur plusieurs invocations continues sans base de données externe. Cependant, avec les fenêtres bascules, vous pouvez maintenir votre état au long des invocations. Cet état contient le résultat global des messages précédemment traités pour la fenêtre actuelle. Votre état peut être d'un maximum de 1 Mo par partition. S'il dépasse cette taille, Lambda met fin précocement à la fenêtre de traitement.

Chaque enregistrement d'un flux appartient à une fenêtre spécifique. La fonction Lambda traitera chaque enregistrement au moins une fois. Toutefois, elle ne garantit pas un seul traitement pour chaque enregistrement. Dans de rares cas, tels que pour la gestion des erreurs, certains enregistrements peuvent être sujet à de multiples traitements. Les dossiers sont toujours traités dans

l'ordre dès la première fois. Si les enregistrements sont traités plusieurs fois, ils peuvent être traités dans le désordre.

Regroupement et traitement

Votre fonction gérée par l'utilisateur est invoquée tant pour l'agrégation que pour le traitement des résultats finaux de celle-ci. Lambda regroupe tous les enregistrements reçus dans la fenêtre. Vous pouvez recevoir ces enregistrements en plusieurs lots, chacun sous forme d'invocation séparée. Chaque invocation reçoit un état. Ainsi, lorsque vous utilisez des fenêtres bascules, votre réponse de fonction Lambda doit contenir une propriété `state`. Si la réponse ne contient pas de propriété `state`, Lambda considère qu'il s'agit d'une invocation ayant échoué. Pour satisfaire à cette condition, votre fonction peut renvoyer un objet `TimeWindowEventResponse` ayant la forme JSON suivante :

Example **`TimeWindowEventResponse`** values

```
{
    "state": {
        "1": 282,
        "2": 715
    },
    "batchItemFailures": []
}
```

> **Note**
>
> Pour les fonctions Java, nous vous recommandons d'utiliser `Map<String, String>` pour représenter l'état.

À la fin de la fenêtre, l'indicateur `isFinalInvokeForWindow` est défini sur `true` pour indiquer qu'il s'agit de l'état final et qu'il est prêt pour le traitement. Après le traitement, la fenêtre et votre invocation final se terminent, puis l'état est supprimé.

À la fin de votre fenêtre, Lambda applique un traitement final pour les actions sur les résultats de l'agrégation. Votre traitement final est invoqué de manière synchrone. Une fois l'invocation réussie, votre fonction contrôle le numéro de séquence et le traitement du flux continue. Si l'invocation échoue, votre fonction Lambda suspend le traitement ultérieur jusqu'à ce que l'invocation soit réussie.

Example DynamoDbTimeWindowEvent

```
{
   "Records":[
      {
         "eventID":"1",
         "eventName":"INSERT",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "NewImage":{
               "Message":{
                  "S":"New item!"
               },
               "Id":{
                  "N":"101"
               }
            },
            "SequenceNumber":"111",
            "SizeBytes":26,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      },
      {
         "eventID":"2",
         "eventName":"MODIFY",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "NewImage":{
```

```
            "Message":{
               "S":"This item has changed"
            },
            "Id":{
               "N":"101"
            }
         },
         "OldImage":{
            "Message":{
               "S":"New item!"
            },
            "Id":{
               "N":"101"
            }
         },
         "SequenceNumber":"222",
         "SizeBytes":59,
         "StreamViewType":"NEW_AND_OLD_IMAGES"
      },
      "eventSourceARN":"stream-ARN"
   },
   {
      "eventID":"3",
      "eventName":"REMOVE",
      "eventVersion":"1.0",
      "eventSource":"aws:dynamodb",
      "awsRegion":"us-east-1",
      "dynamodb":{
         "Keys":{
            "Id":{
               "N":"101"
            }
         },
         "OldImage":{
            "Message":{
               "S":"This item has changed"
            },
            "Id":{
               "N":"101"
            }
         },
         "SequenceNumber":"333",
         "SizeBytes":38,
         "StreamViewType":"NEW_AND_OLD_IMAGES"
```

```
         },
         "eventSourceARN":"stream-ARN"
      }
   ],
    "window": {
        "start": "2020-07-30T17:00:00Z",
        "end": "2020-07-30T17:05:00Z"
    },
    "state": {
        "1": "state1"
    },
    "shardId": "shard123456789",
    "eventSourceARN": "stream-ARN",
    "isFinalInvokeForWindow": false,
    "isWindowTerminatedEarly": false
}
```

Configuration

Vous pouvez configurer des fenêtres bascule lorsque vous créez ou mettez à jour un mappage de source d'événement. Pour configurer une fenêtre bascule, précisez la fenêtre en secondes (TumblingWindowInSeconds). L'exemple de commande AWS Command Line Interface (AWS CLI) suivant crée un mappage de source d'événement de streaming dont la fenêtre bascule est de 120 secondes. La fonction Lambda définie pour l'agrégation et le traitement est nommée `tumbling-window-example-function`.

```
aws lambda create-event-source-mapping \
--event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525 \
--function-name tumbling-window-example-function \
--starting-position TRIM_HORIZON \
--tumbling-window-in-seconds 120
```

Lambda détermine les limites des fenêtres bascule en fonction de l'heure à laquelle les enregistrements ont été insérés dans le flux. Tous les enregistrements ont un horodatage approximatif disponible que Lambda utilise pour déterminer des limites.

Les agrégations de fenêtres bascule ne prennent pas en charge le repartitionnement. Quand la partition prend fin, Lambda considère que la fenêtre de traitement est fermée, et les partitions enfants entament leur propre fenêtre de traitement dans un nouvel état.

Les fenêtres bascule prennent complètement en charge les stratégies de nouvelle tentative existantes `maxRetryAttempts` et `maxRecordAge`.

Example Handler.py – Agrégation et traitement

La fonction Python suivante montre comment regrouper et traiter votre état final :

```
def lambda_handler(event, context):
    print('Incoming event: ', event)
    print('Incoming state: ', event['state'])

#Check if this is the end of the window to either aggregate or process.
    if event['isFinalInvokeForWindow']:
        # logic to handle final state of the window
        print('Destination invoke')
    else:
        print('Aggregate invoke')

#Check for early terminations
    if event['isWindowTerminatedEarly']:
        print('Window terminated early')

    #Aggregation logic
    state = event['state']
    for record in event['Records']:
        state[record['dynamodb']['NewImage']['Id']] = state.get(record['dynamodb']
['NewImage']['Id'], 0) + 1

    print('Returning state: ', state)
    return {'state': state}
```

Paramètres Lambda pour les mappages des sources d'événement Amazon DynamoDB

Tous les types de sources d'événement Lambda partagent les mêmes opérations d'API CreateEventSourceMapping et UpdateEventSourceMapping. Cependant, seuls certains paramètres s'appliquent à DynamoDB Streams.

Paramètre	Obligatoire	Par défaut	Remarques
BatchSize	N	100	Maximum : 10 000.

Paramètre	Obligatoire	Par défaut	Remarques
BisectBatchOnFunctionError	N	FAUX	none
DestinationConfig	N	N/A	File d'attente Amazon SQS standard ou destination de rubrique Amazon SNS standard pour les enregistrements ignorés
Activé	N	VRAI	none
EventSourceArn	Y	N/A	ARN du flux de données ou d'un consommateur de flux
FilterCriteria	N	N/A	Contrôle des événements envoyés par Lambda à votre fonction
FunctionName	Y	N/A	none

Paramètre	Obligatoire	Par défaut	Remarques
FunctionResponseTypes	N	N/A	Pour permettre à votre fonction de signaler des échecs spécifiques dans un lot, incluez la valeur `ReportBatchItemFailures` dans `FunctionResponseTypes` . Pour de plus amples informations, consultez Configuration d'une réponse par lots partielle avec DynamoDB et Lambda.
MaximumBatchingWindowInSeconds	N	0	none
MaximumRecordAgeInSeconds	N	-1	-1 signifie infini : les enregistrements qui ont échoué sont réessayés jusqu'à ce que l'enregistrement expire. La limite de conservation des données pour les flux DynamoDB est de 24 heures. Minimum : -1 Maximum : 604 800

Paramètre	Obligatoire	Par défaut	Remarques
MaximumRetryAttempts	N	-1	-1 signifie infini : les registres qui ont échoué sont réessayés jusqu'à ce que le registre expire. Minimum : 0 Maximum : 10 000.
ParallelizationFactor	N	1	Maximum : 10
StartingPosition	O	N/A	TRIM_HORIZON ou LATEST
TumblingWindowInSeconds	N	N/A	Minimum : 0 Maximum : 900

Utilisation du filtrage des événements avec une source d'événement DynamoDB

Vous pouvez utiliser le filtrage d'événements pour contrôler les enregistrements d'un flux ou d'une file d'attente que Lambda envoie à votre fonction. Pour obtenir des informations générales sur le fonctionnement du filtrage des événements, consultez the section called "Filtrage des événements".

Cette section porte sur le filtrage des événements pour les sources d'événement DynamoDB.

> **Note**
>
> Les mappages des sources d'événements DynamoDB prennent uniquement en charge le filtrage sur la clé `dynamodb`.

Rubriques

- Événement DynamoDB

- Filtrage à l'aide des attributs de table
- Filtrage à l'aide d'expressions booléennes
- Utilisation de l'opérateur Exists
- Format JSON pour le filtrage DynamoDB

Événement DynamoDB

Supposons que vous ayez une table DynamoDB avec la clé primaire `CustomerName` et les attributs `AccountManager` et `PaymentTerms`. La figure suivante montre un exemple d'enregistrement provenant du flux de votre table DynamoDB.

```
{
      "eventID": "1",
      "eventVersion": "1.0",
      "dynamodb": {
          "ApproximateCreationDateTime": "1678831218.0",
          "Keys": {
              "CustomerName": {
                  "S": "AnyCompany Industries"
              }
          },
          "NewImage": {
              "AccountManager": {
                  "S": "Pat Candella"
              },
              "PaymentTerms": {
                  "S": "60 days"
              },
              "CustomerName": {
                  "S": "AnyCompany Industries"
              }
          },
          "SequenceNumber": "111",
          "SizeBytes": 26,
          "StreamViewType": "NEW_IMAGE"
      }
  }
```

Pour filtrer sur la base des valeurs de clé et d'attribut de votre table DynamoDB, utilisez la clé `dynamodb` dans l'enregistrement. Les sections suivantes fournissent des exemples de différents types de filtres.

Filtrage à l'aide des clés de table

Supposons que vous vouliez que votre fonction ne traite que les enregistrements dont la clé primaire `CustomerName` est « AnyCompany Industries. » L'objet `FilterCriteria` serait le suivant.

```
{
     "Filters": [
          {
              "Pattern": "{ \"dynamodb\" : { \"Keys\" : { \"CustomerName\" : { \"S\" :
 [ \"AnyCompany Industries\" ] } } } }"
          }
      ]
 }
```

Pour plus de clarté, voici la valeur du `Pattern` de filtre étendu en JSON simple :

```
{
     "dynamodb": {
          "Keys": {
              "CustomerName": {
                  "S": [ "AnyCompany Industries" ]
                  }
              }
          }
 }
```

Vous pouvez ajouter votre filtre à l'aide de la console, d'AWS CLI ou d'un modèle AWS SAM.

Console

Pour ajouter ce filtre à l'aide de la console, suivez les instructions de Attacher des critères de filtre à un mappage de sources d'événements (console) et saisissez la chaîne suivante comme critère de filtre.

```
{ "dynamodb" : { "Keys" : { "CustomerName" : { "S" : [ "AnyCompany
 Industries" ] } } } }
```

AWS CLI

Pour créer un nouveau mappage des sources d'événements avec ces critères de filtrage à l'aide de l'outil AWS Command Line Interface (AWS CLI), exécutez la commande suivante.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"Keys\" :
 { \"CustomerName\" : { \"S\" : [ \"AnyCompany Industries\" ] } } } }"}]}'
```

Pour ajouter ces critères de filtre à un mappage des sources d'événements existant, exécutez la commande suivante.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"Keys\" :
 { \"CustomerName\" : { \"S\" : [ \"AnyCompany Industries\" ] } } } }"}]}'
```

AWS SAM

Pour ajouter ce filtre à l'aide d'AWS SAM, ajoutez l'extrait suivant au modèle YAML de votre source d'événement.

```
FilterCriteria:
   Filters:
     - Pattern: '{ "dynamodb" : { "Keys" : { "CustomerName" : { "S" : [ "AnyCompany
 Industries" ] } } } }'
```

Filtrage à l'aide des attributs de table

Avec DynamoDB, vous pouvez également utiliser les clés `NewImage` et `OldImage` pour filtrer les valeurs d'attributs. Supposons que vous vouliez filtrer les enregistrements où l'attribut `AccountManager` de la dernière image de la table est « Pat Candella » ou « Shirley Rodriguez ». L'objet `FilterCriteria` serait le suivant.

```
{
    "Filters": [
        {
```

```
            "Pattern": "{ \"dynamodb\" : { \"NewImage\" : { \"AccountManager\" : { \"S
\" : [ \"Pat Candella\", \"Shirley Rodriguez\" ] } } } }"
        }
    ]
}
```

Pour plus de clarté, voici la valeur du `Pattern` de filtre étendu en JSON simple :

```
{
    "dynamodb": {
        "NewImage": {
            "AccountManager": {
                "S": [ "Pat Candella", "Shirley Rodriguez" ]
            }
        }
    }
}
```

Vous pouvez ajouter votre filtre à l'aide de la console, d'AWS CLI ou d'un modèle AWS SAM.

Console

Pour ajouter ce filtre à l'aide de la console, suivez les instructions de Attacher des critères de filtre à un mappage de sources d'événements (console) et saisissez la chaîne suivante comme critère de filtre.

```
{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat Candella",
 "Shirley Rodriguez" ] } } } }
```

AWS CLI

Pour créer un nouveau mappage des sources d'événements avec ces critères de filtrage à l'aide de l'outil AWS Command Line Interface (AWS CLI), exécutez la commande suivante.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\", \"Shirley Rodriguez
\" ] } } } }"}]}'
```

Pour ajouter ces critères de filtre à un mappage des sources d'événements existant, exécutez la commande suivante.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\", \"Shirley Rodriguez
\" ] } } } }"}]}'
```

AWS SAM

Pour ajouter ce filtre à l'aide d'AWS SAM, ajoutez l'extrait suivant au modèle YAML de votre source d'événement.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella", "Shirley Rodriguez" ] } } } }'
```

Filtrage à l'aide d'expressions booléennes

Vous pouvez également créer des filtres à l'aide des expressions booléennes AND. Ces expressions peuvent inclure les paramètres de clé et d'attribut de votre table. Supposons que vous souhaitiez filtrer les enregistrements dont la valeur `NewImage` d'`AccountManager` est « Pat Candella » et la valeur `OldImage` est « Terry Whitlock ». L'objet `FilterCriteria` serait le suivant.

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\" : { \"NewImage\" : { \"AccountManager\" : { \"S
\" : [ \"Pat Candella\" ] } } } , \"dynamodb\" : { \"OldImage\" : { \"AccountManager
\" : { \"S\" : [ \"Terry Whitlock\" ] } } } }"
        }
    ]
}
```

Pour plus de clarté, voici la valeur du `Pattern` de filtre étendu en JSON simple :

```
{
```

```
    "dynamodb" : {
        "NewImage" : {
            "AccountManager" : {
                "S" : [
                    "Pat Candella"
                ]
            }
        }
    },
    "dynamodb": {
        "OldImage": {
            "AccountManager": {
                "S": [
                    "Terry Whitlock"
                ]
            }
        }
    }
}
```

Vous pouvez ajouter votre filtre à l'aide de la console, d'AWS CLI ou d'un modèle AWS SAM.

Console

Pour ajouter ce filtre à l'aide de la console, suivez les instructions de Attacher des critères de filtre à un mappage de sources d'événements (console) et saisissez la chaîne suivante comme critère de filtre.

```
{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella" ] } } } , "dynamodb" : { "OldImage" : { "AccountManager" : { "S" :
 [ "Terry Whitlock" ] } } } }
```

AWS CLI

Pour créer un nouveau mappage des sources d'événements avec ces critères de filtrage à l'aide de l'outil AWS Command Line Interface (AWS CLI), exécutez la commande suivante.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\" ] } } } , \"dynamodb\" :
```

```
 { \"OldImage\" : { \"AccountManager\" : { \"S\" : [ \"Terry Whitlock\" ] } } } }
 "}]}'
```

Pour ajouter ces critères de filtre à un mappage des sources d'événements existant, exécutez la commande suivante.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\" ] } } } , \"dynamodb\" :
 { \"OldImage\" : { \"AccountManager\" : { \"S\" : [ \"Terry Whitlock\" ] } } } }
 "}]}'
```

AWS SAM

Pour ajouter ce filtre à l'aide d'AWS SAM, ajoutez l'extrait suivant au modèle YAML de votre source d'événement.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella" ] } } } , "dynamodb" : { "OldImage" : { "AccountManager" : { "S" :
 [ "Terry Whitlock" ] } } } }'
```

> **Note**
>
> Le filtrage d'événements DynamoDB ne prend pas en charge l'utilisation d'opérateurs numériques (égalité numérique et plage numérique). Même si les éléments de votre table sont stockés sous forme de nombres, ces paramètres sont convertis en chaînes dans l'objet d'enregistrement JSON.

Utilisation de l'opérateur Exists

En raison de la structure des objets d'événements JSON de DynamoDB, l'utilisation de l'opérateur Exists nécessite une attention particulière. L'opérateur Exists ne fonctionne que sur les nœuds terminaux dans l'événement JSON. Par conséquent, si votre modèle de filtre utilise Exists pour tester un nœud intermédiaire, il ne fonctionnera pas. Observez l'élément de table DynamoDB suivant :

```
{
  "UserID": {"S": "12345"},
  "Name": {"S": "John Doe"},
  "Organizations": {"L": [
      {"S":"Sales"},
      {"S":"Marketing"},
      {"S":"Support"}
    ]
  }
}
```

Vous pourriez avoir besoin de créer un modèle de filtre comme le suivant pour tester les événements contenant `"Organizations"` :

```
{ "dynamodb" : { "NewImage" : { "Organizations" : [ { "exists": true } ] } } }
```

Cependant, ce modèle de filtre ne renverra jamais de correspondance, car `"Organizations"` n'est pas un nœud terminal. L'exemple suivant montre comment utiliser correctement l'opérateur Exists pour construire le modèle de filtre souhaité :

```
{ "dynamodb" : { "NewImage" : {"Organizations": {"L": {"S": [ {"exists":
 true } ] } } } } }
```

Format JSON pour le filtrage DynamoDB

Pour filtrer correctement les événements provenant de sources DynamoDB, le champ de données et vos critères de filtre pour le champ de données (`dynamodb`) doivent être au format JSON valide. Si l'un ou l'autre des champs n'est pas dans un format JSON valide, Lambda rejette le message ou lance une exception. Le tableau suivant résume le comportement spécifique :

Format des données entrantes	Pas de modèle de filtre pour les propriétés des données	Action obtenue.
JSON valide	JSON valide	Lambda filtre en fonction de vos critères de filtre.
JSON valide	Pas de modèle de filtre pour les propriétés des données	Lambda filtre (uniquement selon les autres propriétés de

Format des données entrantes	Pas de modèle de filtre pour les propriétés des données	Action obtenue.
		métadonnées) en fonction de vos critères de filtre.
JSON valide	Non JSON	Lambda lance une exception au moment de la création ou de la mise à jour du mappage de sources d'événements. Le modèle de filtre des propriétés de données doit être au format JSON valide.
Non JSON	JSON valide	Lambda rejette l'enregis trement.
Non JSON	Pas de modèle de filtre pour les propriétés des données	Lambda filtre (uniquement selon les autres propriétés de métadonnées) en fonction de vos critères de filtre.
Non JSON	Non JSON	Lambda lance une exception au moment de la création ou de la mise à jour du mappage de sources d'événements. Le modèle de filtre des propriétés de données doit être au format JSON valide.

Tutoriel : Utilisation AWS Lambda avec les flux Amazon DynamoDB

Dans ce didacticiel, vous allez créer une fonction Lambda afin d'utiliser des événements d'un flux Amazon DynamoDB.

Conditions préalables

Installez le AWS Command Line Interface

Si vous ne l'avez pas encore installé AWS Command Line Interface, suivez les étapes décrites dans la section Installation ou mise à jour de la dernière version du AWS CLI pour l'installer.

Ce tutoriel nécessite un terminal de ligne de commande ou un shell pour exécuter les commandes. Sous Linux et macOS, utilisez votre gestionnaire de shell et de package préféré.

> **Note**
>
> Sous Windows, certaines commandes CLI Bash que vous utilisez couramment avec Lambda (par exemple `zip`) ne sont pas prises en charge par les terminaux intégrés du système d'exploitation. Installez le sous-système Windows pour Linux afin d'obtenir une version intégrée à Windows d'Ubuntu et Bash.

Créer le rôle d'exécution

Créez le rôle d'exécution qui autorise votre fonction à accéder aux AWS ressources.

Pour créer un rôle d'exécution

1. Ouvrez la page Roles (Rôles) dans la console IAM.
2. Sélectionnez Créer un rôle.
3. Créez un rôle avec les propriétés suivantes :
   - Entité de confiance – Lambda.
   - Autorisations — DBExecutionRôle AWSLambda Dynamo.
   - Nom de rôle – **`lambda-dynamodb-role`**.

Le DBExecutionrôle AWSLambda Dynamo dispose des autorisations dont la fonction a besoin pour lire des éléments depuis DynamoDB et écrire des journaux dans des journaux. CloudWatch

Créer la fonction

Créez une fonction Lambda qui traite vos événements DynamoDB. Le code de fonction écrit certaines des données d'événements entrants dans CloudWatch Logs.

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement DynamoDB avec Lambda en utilisant .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public void FunctionHandler(DynamoDBEvent dynamoEvent, ILambdaContext
 context)
    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");

        foreach (var record in dynamoEvent.Records)
        {
            context.Logger.LogInformation($"Event ID: {record.EventID}");
            context.Logger.LogInformation($"Event Name: {record.EventName}");

            context.Logger.LogInformation(JsonSerializer.Serialize(record));
        }

        context.Logger.LogInformation("Stream processing complete.");
```

```
    }
}
```

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement DynamoDB avec Lambda en utilisant Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-lambda-go/events"
 "fmt"
)

func HandleRequest(ctx context.Context, event events.DynamoDBEvent) (*string,
 error) {
 if len(event.Records) == 0 {
  return nil, fmt.Errorf("received empty event")
 }

 for _, record := range event.Records {
   LogDynamoDBRecord(record)
 }

 message := fmt.Sprintf("Records processed: %d", len(event.Records))
 return &message, nil
}

func main() {
```

```
 lambda.Start(HandleRequest)
}

func LogDynamoDBRecord(record events.DynamoDBEventRecord){
 fmt.Println(record.EventID)
 fmt.Println(record.EventName)
 fmt.Printf("%+v\n", record.Change)
}
```

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement DynamoDB avec Lambda en utilisant Java.

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import
 com.amazonaws.services.lambda.runtime.events.DynamodbEvent.DynamodbStreamRecord;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class example implements RequestHandler<DynamodbEvent, Void> {

    private static final Gson GSON = new
 GsonBuilder().setPrettyPrinting().create();

    @Override
    public Void handleRequest(DynamodbEvent event, Context context) {
        System.out.println(GSON.toJson(event));
        event.getRecords().forEach(this::logDynamoDBRecord);
        return null;
    }

    private void logDynamoDBRecord(DynamodbStreamRecord record) {
```

```
        System.out.println(record.getEventID());
        System.out.println(record.getEventName());
        System.out.println("DynamoDB Record: " +
 GSON.toJson(record.getDynamodb()));
    }
}
```

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement DynamoDB avec Lambda en utilisant. JavaScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
};

const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

Consommation d'un événement DynamoDB avec Lambda en utilisant. TypeScript

```
export const handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
```

```
    });
}
const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement DynamoDB avec Lambda en utilisant PHP.

```
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\DynamoDb\DynamoDbHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends DynamoDbHandler
{
    private StderrLogger $logger;

    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
```

```
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleDynamoDb(DynamoDbEvent $event, Context $context): void
    {
        $this->logger->info("Processing DynamoDb table items");
        $records = $event->getRecords();

        foreach ($records as $record) {
            $eventName = $record->getEventName();
            $keys = $record->getKeys();
            $old = $record->getOldImage();
            $new = $record->getNewImage();

            $this->logger->info("Event Name:".$eventName."\n");
            $this->logger->info("Keys:". json_encode($keys)."\n");
            $this->logger->info("Old Image:". json_encode($old)."\n");
            $this->logger->info("New Image:". json_encode($new));

            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }

        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords items");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement DynamoDB avec Lambda en utilisant Python.

```
import json

def lambda_handler(event, context):
    print(json.dumps(event, indent=2))

    for record in event['Records']:
        log_dynamodb_record(record)

def log_dynamodb_record(record):
    print(record['eventID'])
    print(record['eventName'])
    print(f"DynamoDB Record: {json.dumps(record['dynamodb'])}")
```

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement DynamoDB avec Lambda en utilisant Ruby.

```
def lambda_handler(event:, context:)
    return 'received empty event' if event['Records'].empty?

    event['Records'].each do |record|
      log_dynamodb_record(record)
    end

    "Records processed: #{event['Records'].length}"
  end

  def log_dynamodb_record(record)
```

```
    puts record['eventID']
    puts record['eventName']
    puts "DynamoDB Record: #{JSON.generate(record['dynamodb'])}"
  end
```

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement DynamoDB avec Lambda en utilisant Rust.

```
use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord},
   };


// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
//tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<Event>) ->Result<(), Error> {

    let records = &event.payload.records;
    tracing::info!("event payload: {:?}",records);
    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }
```

```
    for record in records{
        log_dynamo_dbrecord(record);
    }

    tracing::info!("Dynamo db records processed");

    // Prepare the response
    Ok(())

}

fn log_dynamo_dbrecord(record: &EventRecord)-> Result<(), Error>{
    tracing::info!("EventId: {}", record.event_id);
    tracing::info!("EventName: {}", record.event_name);
    tracing::info!("DynamoDB Record: {:?}", record.change );
    Ok(())

}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())

}
```

Pour créer la fonction

1. Copiez l'exemple de code dans un fichier nommé `example.js`.
2. Créez un package de déploiement.

```
zip function.zip example.js
```

3. Créez une fonction Lambda à l'aide de la commande `create-function`.

```
aws lambda create-function --function-name ProcessDynamoDBRecords \
    --zip-file fileb://function.zip --handler example.handler --runtime nodejs24.x
 \
    --role arn:aws:iam::111122223333:role/lambda-dynamodb-role
```

Test de la fonction Lambda

Au cours de cette étape, vous appelez votre fonction Lambda manuellement à l'aide de la commande `invoke` AWS Lambda CLI et de l'exemple d'événement DynamoDB suivant. Copiez le code suivant dans un fichier nommé `input.txt`.

Example input.txt

```
{
   "Records":[
      {
         "eventID":"1",
         "eventName":"INSERT",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "NewImage":{
               "Message":{
                  "S":"New item!"
               },
               "Id":{
                  "N":"101"
               }
            },
            "SequenceNumber":"111",
            "SizeBytes":26,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
```

```
    },
    {
       "eventID":"2",
       "eventName":"MODIFY",
       "eventVersion":"1.0",
       "eventSource":"aws:dynamodb",
       "awsRegion":"us-east-1",
       "dynamodb":{
          "Keys":{
             "Id":{
                "N":"101"
             }
          },
          "NewImage":{
             "Message":{
                "S":"This item has changed"
             },
             "Id":{
                "N":"101"
             }
          },
          "OldImage":{
             "Message":{
                "S":"New item!"
             },
             "Id":{
                "N":"101"
             }
          },
          "SequenceNumber":"222",
          "SizeBytes":59,
          "StreamViewType":"NEW_AND_OLD_IMAGES"
       },
       "eventSourceARN":"stream-ARN"
    },
    {
       "eventID":"3",
       "eventName":"REMOVE",
       "eventVersion":"1.0",
       "eventSource":"aws:dynamodb",
       "awsRegion":"us-east-1",
       "dynamodb":{
          "Keys":{
             "Id":{
```

```
                    "N":"101"
                }
            },
            "OldImage":{
                "Message":{
                    "S":"This item has changed"
                },
                "Id":{
                    "N":"101"
                }
            },
            "SequenceNumber":"333",
            "SizeBytes":38,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      }
   ]
}
```

Exécutez la commande suivante `invoke`.

```
aws lambda invoke --function-name ProcessDynamoDBRecords \
    --cli-binary-format raw-in-base64-out \
    --payload file://input.txt outputfile.txt
```

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales prises en charge par l'AWS CLI dans le Guide de l'utilisateur AWS Command Line Interface version 2.

La fonction renvoie la chaîne `message` dans le corps de la réponse.

Vérifiez la sortie dans le fichier `outputfile.txt`.

Créer une table DynamoDB avec un flux activé

Créez une table Amazon DynamoDB avec un flux activé.

Pour créer une table DynamoDB

1. Ouvrez la console DynamoDB.

2. Choisissez Créer un tableau.
3. Créez une table avec les paramètres suivants.
   - Nom de la table – **`lambda-dynamodb-stream`**
   - Clé primaire – **`id`** (chaîne)
4. Choisissez Create (Créer).

Pour activer les flux

1. Ouvrez la console DynamoDB.
2. Choisissez Tables.
3. Choisissez la table lambda-dynamodb-stream.
4. Sous la section Exports and streams (Exportations et flux), choisissez DynamoDB stream details (Détails du flux DynamoDB).
5. Choisissez Activer.
6. Pour Type de vue, sélectionnez Attributs de clé uniquement.
7. Choisissez Activer le streaming.

Écrivez l'ARN du flux. Vous en aurez besoin à l'étape suivante pour associer le flux à votre fonction Lambda. Pour plus d'informations sur l'activation des flux, consultez Capture d'activité Table avec DynamoDB Streams.

Ajouter une source d'événement dans AWS Lambda

Créez un mappage de source d'événement dans AWS Lambda. Ce mappage de source d'événement associe le flux DynamoDB avec votre fonction Lambda. Après avoir créé ce mappage des sources d'événements, AWS Lambda commence à interroger le flux.

Exécutez la commande suivante AWS CLI `create-event-source-mapping`. Une fois la commande exécutée, notez l'UUID. Vous aurez besoin de l'UUID pour faire référence au mappage de source d'événement dans les commandes (par exemple, lors de la suppression du mappage).

```
aws lambda create-event-source-mapping --function-name ProcessDynamoDBRecords \
    --batch-size 100 --starting-position LATEST --event-source DynamoDB-stream-arn
```

Cela a pour effet de créer un mappage entre le flux DynamoDB spécifié et la fonction Lambda. Vous pouvez associer un flux diffuser à plusieurs fonctions Lambda, et associer la même fonction Lambda à plusieurs flux. Toutefois, les fonctions Lambda partagent le débit de lecture du flux qu'elles partagent.

Pour obtenir la liste des mappages de source d'événement, exécutez la commande suivante.

```
aws lambda list-event-source-mappings
```

Cette liste renvoie tous les mappages de source d'événement que vous avez créés et indique la valeur `LastProcessingResult` pour chacun d'eux, entre autres. Ce champ est utilisé pour fournir un message d'information en cas de problème. Des valeurs telles que `No records processed` (indique que le sondage n' AWS Lambda a pas commencé ou qu'il n'y a aucun enregistrement dans le flux) et OK (indique que les enregistrements du flux AWS Lambda ont été lus avec succès et que votre fonction Lambda a été invoquée) indiquent qu'il n'y a aucun problème. Dans le cas contraire, vous recevez un message d'erreur.

Si vous avez un grand nombre de mappages de source d'événement, utilisez le paramètre du nom de la fonction pour affiner les résultats.

```
aws lambda list-event-source-mappings --function-name ProcessDynamoDBRecords
```

Tester la configuration

Testez l' end-to-endexpérience. A mesure que vous mettez la table à jour, DynamoDB écrit les enregistrements d'événement dans le flux. Quand AWS Lambda interroge le flux, il y détecte les nouveaux enregistrements et invoque pour vous la fonction Lambda en transmettant les événements à la fonction.

1. Dans la console DynamoDB, vous pouvez ajouter, mettre à jour et supprimer des éléments dans la table. DynamoDB écrit des enregistrements de ces actions dans le flux.
2. AWS Lambda interroge le flux et lorsqu'il détecte des mises à jour du flux, il invoque votre fonction Lambda en transmettant les données d'événements qu'il trouve dans le flux.
3. Votre fonction s'exécute et crée des journaux sur Amazon CloudWatch. Vous pouvez vérifier les journaux signalés dans la CloudWatch console Amazon.

Étapes suivantes

Ce didacticiel vous a montré les principes de base du traitement des événements de flux DynamoDB avec Lambda. Pour les charges de travail de production, envisagez de mettre en œuvre une logique de réponse partielle de lot afin de gérer plus efficacement les échecs d'enregistrement individuels. L'utilitaire de traitement par lots de Powertools for AWS Lambda est disponible en Python TypeScript, .NET et Java et fournit une solution robuste pour cela, en gérant automatiquement la complexité des réponses partielles par lots et en réduisant le nombre de tentatives pour les enregistrements traités avec succès.

Nettoyage de vos ressources

Vous pouvez maintenant supprimer les ressources que vous avez créées pour ce didacticiel, sauf si vous souhaitez les conserver. En supprimant AWS les ressources que vous n'utilisez plus, vous évitez des frais inutiles pour votre Compte AWS.

Pour supprimer la fonction Lambda

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction que vous avez créée.
3. Sélectionnez Actions, Supprimer.
4. Saisissez **`confirm`** dans la zone de saisie de texte et choisissez Delete (Supprimer).

Pour supprimer le rôle d'exécution

1. Ouvrez la page Roles (Rôles) de la console IAM.
2. Sélectionnez le rôle d'exécution que vous avez créé.
3. Sélectionnez Delete (Supprimer).
4. Saisissez le nom du rôle dans le champ de saisie de texte et choisissez Delete (Supprimer).

Pour supprimer la table DynamoDB

1. Ouvrez la page Tables (Tables) de la console DynamoDB.
2. Sélectionnez la table que vous avez créée.
3. Choisissez Supprimer.
4. Saisissez **`delete`** dans la zone de texte.

5. Choisissez Supprimer la table.

Traiter les événements du cycle de vie Amazon EC2 avec une fonction Lambda

Vous pouvez utiliser AWS Lambda pour traiter des événements de cycle de vie d'Amazon Elastic Compute Cloud et gérer des ressources Amazon EC2. Amazon EC2 envoie des événements à Amazon EventBridge (CloudWatch Events) pour des événements de cycle de vie, par exemple, quand une instance change d'état, quand un instantané de volume Amazon Elastic Block Store est terminé ou quand une instance Spot est planifiée pour être arrêtée. Vous configurez EventBridge (CloudWatch Events) pour transférer ces événements à une fonction Lambda en vue de leur traitement.

EventBridge (CloudWatch Events) appelle la fonction Lambda de manière asynchrone avec le document d'événement d'Amazon EC2.

Example événement du cycle de vie d'une instance

```
{
    "version": "0",
    "id": "b6ba298a-7732-2226-xmpl-976312c1a050",
    "detail-type": "EC2 Instance State-change Notification",
    "source": "aws.ec2",
    "account": "111122223333",
    "time": "2019-10-02T17:59:30Z",
    "region": "us-east-1",
    "resources": [
        "arn:aws:ec2:us-east-1:111122223333:instance/i-0c314xmplcd5b8173"
    ],
    "detail": {
        "instance-id": "i-0c314xmplcd5b8173",
        "state": "running"
    }
}
```

Pour plus d'informations sur la configuration des événements, consultez Invocation d'une fonction Lambda dans une planification. Pour obtenir un exemple de fonction qui traite les notifications d'instantanés Amazon EBS, consultez EventBridge Scheduler for Amazon EBS.

Vous pouvez également utiliser le kit SDK AWS pour gérer les instances et d'autres ressources avec l'API Amazon EC2.

Octroi d'autorisations à EventBridge (CloudWatch Events)

Pour traiter les événements de cycle de vie d'Amazon EC2, EventBridge (CloudWatch Events) doit être autorisé à appeler votre fonction. Cette autorisation provient de la stratégie basée sur les ressourcesde la fonction. Si vous utilisez la console EventBridge (CloudWatch Events) pour configurer un déclencheur d'événement, la console met à jour la stratégie basée sur une ressource en votre nom. Sinon, ajoutez une instruction comme suit :

Example Déclaration de stratégie basée sur une ressource pour les notifications de cycle de vie Amazon EC2

```
{
  "Sid": "ec2-events",
  "Effect": "Allow",
  "Principal": {
    "Service": "events.amazonaws.com"
  },
  "Action": "lambda:InvokeFunction",
  "Resource": "arn:aws:lambda:us-east-1:12456789012:function:my-function",
  "Condition": {
    "ArnLike": {
      "AWS:SourceArn": "arn:aws:events:us-east-1:12456789012:rule/*"
    }
  }
}
```

Pour ajouter une instruction, utilisez la commande `add-permission` AWS CLI.

```
aws lambda add-permission --action lambda:InvokeFunction --statement-id ec2-events \
--principal events.amazonaws.com --function-name my-function --source-arn
 'arn:aws:events:us-east-1:12456789012:rule/*'
```

Si votre fonction utilise le kit SDK AWS pour gérer les ressources Amazon EC2, ajoutez des autorisations Amazon EC2 au rôle d'exécution de la fonction.

Traitet les demandes d’Application Load Balancer avec Lambda

Vous pouvez utiliser une fonction Lambda pour traiter les demandes d’un Application Load Balancer. Elastic Load Balancing prend désormais en charge les fonctions Lambda en tant que cibles pour un Application Load Balancer. Utilisez les règles de l’équilibreur de charge pour acheminer les demandes HTTP vers une fonction, selon le chemin d’accès ou les valeurs des en-têtes. Traitez la demande et renvoyez une réponse HTTP à partir de votre fonction Lambda.

Elastic Load Balancing appelle votre fonction Lambda de façon synchrone avec un événement qui contient le corps et les métadonnées de la demande.

Example Evénement de demande d’Application Load Balancer

```
{
    "requestContext": {
        "elb": {
            "targetGroupArn": "arn:aws:elasticloadbalancing:us-
east-1:123456789012:targetgroup/lambda-279XGJDqGZ5rsrHC2Fjr/49e9d65c45c6791a"
        }
    },
    "httpMethod": "GET",
    "path": "/lambda",
    "queryStringParameters": {
        "query": "1234ABCD"
    },
    "headers": {
        "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,image/apng,*/*;q=0.8",
        "accept-encoding": "gzip",
        "accept-language": "en-US,en;q=0.9",
        "connection": "keep-alive",
        "host": "lambda-alb-123578498.us-east-1.elb.amazonaws.com",
        "upgrade-insecure-requests": "1",
        "user-agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
 (KHTML, like Gecko) Chrome/71.0.3578.98 Safari/537.36",
        "x-amzn-trace-id": "Root=1-5c536348-3d683b8b04734faae651f476",
        "x-forwarded-for": "72.12.164.125",
        "x-forwarded-port": "80",
        "x-forwarded-proto": "http",
        "x-imforwards": "20"
    },
    "body": "",
```

```
    "isBase64Encoded": False
}
```

Votre fonction traite l'événement et renvoie un document de réponse à l'équilibreur de charge en JSON. Elastic Load Balancing convertit le document en réponse de succès ou d'erreur HTTP, et la renvoie à l'utilisateur.

Example format du document de réponse

```
{
    "statusCode": 200,
    "statusDescription": "200 OK",
    "isBase64Encoded": False,
    "headers": {
        "Content-Type": "text/html"
    },
    "body": "<h1>Hello from Lambda!</h1>"
}
```

Pour configurer un Application Load Balancer comme déclencheur de fonction, accordez à Elastic Load Balancing l'autorisation d'exécuter la fonction, créez un groupe cible qui achemine les demandes vers la fonction, et ajoutez à l'équilibreur de charge une règle qui envoie les demandes au groupe cible.

Utilisez la commande `add-permission` pour ajouter une instruction d'autorisation à la stratégie basée sur les ressources de votre fonction.

```
aws lambda add-permission --function-name alb-function \
--statement-id load-balancer --action "lambda:InvokeFunction" \
--principal elasticloadbalancing.amazonaws.com
```

Vous devriez voir la sortie suivante:

```
{
    "Statement": "{\"Sid\":\"load-balancer\",\"Effect\":\"Allow\",\"Principal\":
{\"Service\":\"elasticloadbalancing.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction
\",\"Resource\":\"arn:aws:lambda:us-west-2:123456789012:function:alb-function\"}"
}
```

Pour obtenir des instructions sur la configuration de l'écouteur de l'équilibreur de charge d'application et du groupe cible, consultez Fonctions Lambda en tant que cibles dans le Guide de l'utilisateur des Application Load Balancers.

Gestionnaire d'événements de Powertools pour Lambda AWS

Le gestionnaire d'événements du kit d'outils Powertools for AWS Lambda fournit le routage, le middleware, la configuration CORS, la génération de spécifications OpenAPI, la validation des demandes, la gestion des erreurs et d'autres fonctionnalités utiles lors de l'écriture de fonctions Lambda invoquées par un Application Load Balancer. L'utilitaire de gestionnaire d'événements est disponible pour Python. Pour plus d'informations, consultez l'API REST du gestionnaire d'événements dans la documentation de Powertools for AWS Lambda (Python).

Python

```
import requests
from requests import Response

from aws_lambda_powertools import Logger, Tracer
from aws_lambda_powertools.event_handler import ALBResolver
from aws_lambda_powertools.logging import correlation_paths
from aws_lambda_powertools.utilities.typing import LambdaContext

tracer = Tracer()
logger = Logger()
app = ALBResolver()


@app.get("/todos")
@tracer.capture_method
def get_todos():
    todos: Response = requests.get("https://jsonplaceholder.typicode.com/todos")
    todos.raise_for_status()

    # for brevity, we'll limit to the first 10 only
    return {"todos": todos.json()[:10]}


# You can continue to use other utilities just as before
@logger.inject_lambda_context(correlation_id_path=correlation_paths.APPLICATION_LOAD_BALANCER)
@tracer.capture_lambda_handler
def lambda_handler(event: dict, context: LambdaContext) -> dict:
```

```
    return app.resolve(event, context)
```

Invocation d'une fonction Lambda dans une planification

Le planificateur Amazon EventBridge est un planificateur sans serveur qui vous permet de créer, d'exécuter et de gérer des tâches à partir d'un service central et géré. Avec le planificateur EventBridge, vous pouvez créer des planifications à l'aide d'expressions cron et rate pour les modèles récurrents, voire configurer des invocations ponctuelles. Vous pouvez configurer des fenêtres temporelles flexibles pour la livraison, définir des limites de nouvelles tentatives, ainsi que la durée de rétention maximale pour les événements non traités.

Lorsque vous configurez le planificateur EventBridge avec Lambda, le planificateur EventBridge invoque votre fonction Lambda de manière asynchrone. Cette page explique comment utiliser le planificateur EventBridge pour invoquer une fonction Lambda dans une planification.

Configurer le rôle d'exécution

Lorsque vous créez une planification, le planificateur EventBridge doit être autorisé à invoquer son opération d'API cible en votre nom. Vous accordez ces autorisations au planificateur EventBridge à l'aide d'un rôle d'exécution. La politique d'autorisation que vous associez au rôle d'exécution de votre planification définit les autorisations requises. Ces autorisations dépendent de l'API cible que vous souhaitez que le planificateur EventBridge invoque.

Lorsque vous utilisez la console du planificateur EventBridge pour créer une planification, comme dans la procédure suivante, le planificateur EventBridge définit automatiquement un rôle d'exécution en fonction de la cible que vous avez sélectionnée. Si vous souhaitez créer une planification à l'aide de l'un des kits SDK du planificateur EventBridge, de la AWS CLI ou de CloudFormation, vous devez disposer d'un rôle d'exécution existant qui accorde les autorisations dont le planificateur EventBridge a besoin pour invoquer une cible. Pour plus d'informations sur la configuration manuelle d'un rôle d'exécution pour votre planification, voir Configuration d'un rôle d'exécution dans le Guide de l'utilisateur du planificateur EventBridge.

Créer une planification

Pour créer une planification à l'aide de la console

1. Ouvrez la console du planificateur Amazon EventBridge à l'adresse https://console.aws.amazon.com/scheduler/home.
2. Sur la page Planifications, choisissez Créer une planification.

3. Sur la page Spécifier le détail de la planification, dans la section Nom et description de la planification, procédez comme suit :

   a. Pour Nom de la planification, saisissez un nom à attribuer à votre planification. Par exemple, **`MyTestSchedule`**.

   b. (Facultatif) Dans le champ Description, saisissez une description de la planification. Par exemple, **`My first schedule`**.

   c. Pour Groupe de planifications, choisissez un groupe de planifications dans la liste déroulante. Si vous n'avez pas de groupe, choisissez par défaut. Pour créer un groupe de planifications, choisissez Crée votre propre planification.

   Vous utilisez des groupes de planifications pour leur ajouter des balises.

4. • Choisissez vos options de planification.

Occurrence	Faites ceci...	
Planification ponctuelle Une planification ponctuelle n'invoque un objectif qu'une seule fois à la date et à l'heure que vous indiquez.	Pour Date et heure, procédez comme suit : • Entrez une date valide au format `YYYY/MM/DD` . • Entrez un horodatage au format `hh:mm` de 24 heures. • Dans le champ Fuseau horaire, choisissez le fuseau horaire.	
Planification récurrente Une planification récurrente invoque un objectif à un taux que vous spécifiez à l'aide d'une expression cron ou d'une expression rate.	a. Pour Schedule type (Planifier le type), effectuez l'une des étapes suivantes : • Pour utiliser une expression cron afin de définir la planification, choisissez	

Occurrence	Faites ceci...	
	Planification basée sur cron et entrez l'expression cron. • Pour utiliser une expression de rythme pour définir la planification, choisissez Planification basée sur le rythme. Pour plus d'informations sur les expressions cron et rate, consultez Types de planifications sur le planificateur EventBridge dans le Guide de l'utilisateur du planificateur Amazon EventBridge. b. Pour Fenêtre temporelle flexible, choisissez Désactivé pour désactiver cette option ou choisir l'une des fenêtres temporelles prédéfinies. Par exemple, si vous choisissez15 minutes et que vous définissez une planification récurrente pour invoquer son objectif une fois par heure, la planification s'exécute dans les 15	

Occurrence	Faites ceci...	
	minutes suivant le début de chaque heure.	

5. (Facultatif) Si vous avez choisi Planification récurrente à l'étape précédente, dans la section Délai, procédez comme suit :

   a. Dans le champ Fuseau horaire, choisissez un fuseau horaire.

   b. Pour Date et heure de début, entrez une date valide au format `YYYY/MM/DD`, puis spécifiez un horodatage au format `hh:mm` de 24 heures.

   c. Pour Date et heure de fin, entrez une date valide au format `YYYY/MM/DD`, puis spécifiez un horodatage au format `hh:mm` de 24 heures.

6. Choisissez Suivant.

7. Sur la page Sélectionner la cible, choisissez l'opération d'API AWS invoquée par le planificateur EventBridge :

   a. Sélectionnez Invoquer AWS Lambda.

   b. Dans la section Invoquer, sélectionnez une fonction ou choisissez Créer une fonction Lambda.

   c. (Facultatif) Entrez une charge utile JSON. Si vous n'entrez aucune charge utile, le planificateur EventBridge utilise un événement vide pour invoquer la fonction.

8. Choisissez Suivant.

9. Sur la page Settings (Paramètres), procédez comme suit :

   a. Pour activer la planification, sous État de la planification, activez Activer la planification.

   b. Pour configurer une stratégie de nouvelles tentatives pour votre planification, sous Politique de nouvelle tentative et file d'attente de lettres mortes (DLQ), procédez comme suit :

      - Activez Réessayer.
      - Pour Âge maximum de l'événement, entrez le nombre maximum d'heures et de minutes de conservation d'un événement non traité par le planificateur EventBridge.
      - La durée maximale est 24 heures.
      - Pour Nombre maximum de tentatives, entrez le nombre maximum de tentatives de renvoi d'une erreur par le planificateur EventBridge.

La valeur maximale est 185 nouvelles tentatives.

Avec les stratégies de nouvelles tentatives, si une planification ne parvient pas à invoquer sa cible, le planificateur EventBridge la réexécute. Si elle est configurée, vous devez définir la durée de rétention maximale et les nouvelles tentatives pour la planification.

c. Choisissez où le planificateur EventBridge stocke les événements non livrés.

Option File d'attente de lettres mortes (DLQ)	Faites ceci...
Ne stockez pas	Sélectionnez Aucun.
Stocker l'événement dans le même Compte AWS où vous créez la planification	a. Choisissez Sélection nez une file d'attente Amazon SQS dans mon Compte AWS en tant que DLQ. b. Choisissez l'Amazon Resource Name (ARN) de la file d'attente Amazon SQS.
Stocker l'événement dans un autre Compte AWS que celui où vous créez la planification	a. Choisissez Spécifier une file d'attente Amazon SQS dans un autre Comptes AWS en tant que DLQ. b. Entrez l'Amazon Resource Name (ARN) de la file d'attente Amazon SQS.

d. Pour utiliser une clé gérée par le client afin de chiffrer votre entrée cible, sous Chiffrement, choisissez Personnaliser les paramètres de chiffrement (avancé).

Si vous choisissez cette option, entrez un ARN de clé KMS existant ou choisissez Créez un AWS KMS keypour accéder à la console AWS KMS. Pour plus d'informations sur la façon dont le planificateur EventBridge chiffre vos données au repos, voir Chiffrement au repos dans le Guide de l'utilisateur du planificateur Amazon EventBridge.

e. Pour que le planificateur EventBridge crée un rôle d'exécution pour vous, choisissez Créer un rôle pour cette planification. Ensuite, saisissez un nom pour Nom du rôle. Si vous choisissez cette option, le planificateur EventBridge associe au rôle les autorisations requises pour votre cible modélisée.

10. Choisissez Suivant.
11. Sur la page Examiner et créer une planification, examinez les détails de votre planification. Dans chaque section, choisissez Modifier pour revenir à cette étape et modifier ses détails.
12. Choisissez Créer une planification.

    Vous pouvez consulter la liste de vos planifications nouvelles et existantes sur la page Planifications. Sous la colonne État, vérifiez que votre nouvelle planification est activée.

Pour confirmer que le planificateur EventBridge a invoqué la fonction, consultez les Amazon CloudWatch Logs de la fonction.

Ressources connexes

Pour de plus amples informations sur le planificateur EventBridge, veuillez consulter les ressources suivantes :

- Guide de l'utilisateur du planificateur EventBridge
- Référence de l'API du planificateur EventBridge
- Tarification du planificateur EventBridge

Utilisation de AWS Lambda avec AWS IoT

AWS IoT fournit une communication sécurisée entre les périphériques connectés à Internet (tels que les capteurs) et le Cloud AWS. Vous pouvez ainsi collecter les données de télémétrie de plusieurs périphériques, les stocker et les analyser.

Vous pouvez créer des règles AWS IoT permettant à vos appareils d'interagir avec les Services AWS. Le moteur de règles AWS IoT fournit un langage SQL pour sélectionner des données dans des charges utiles de messages, et les envoyer à d'autres services tels que Amazon S3, Amazon DynamoDB et AWS Lambda. Vous définissez une règle pour appeler une fonction Lambda lorsque vous souhaitez appeler un autre service AWS ou un service tiers.

Quand un message IoT entrant déclenche la règle, AWS IoT appelle votre fonction Lambda de manière asynchrone, et lui transmet les données du message IoT à la fonction.

L'exemple suivant montre une mesure de l'humidité à partir d'un capteur de serre. Les valeurs row et pos identifient l'emplacement du capteur. Cet exemple d'événement est basé sur le type de serre indiqué dans les didacticiels sur les règles AWS IoT.

Example AWS IoTÉvénement de message

```
{
    "row" : "10",
    "pos" : "23",
    "moisture" : "75"
}
```

Pour l'appel asynchrone, Lambda met le message en file d'attente et fait une nouvelle tentative si votre fonction renvoie une erreur. Configurez votre fonction avec une destination pour conserver les événements que votre fonction n'a pas pu traiter.

Vous devez accorder au service AWS IoT l'autorisation d'appeler votre fonction Lambda. Utilisez la commande `add-permission` pour ajouter une instruction d'autorisation à la stratégie basée sur les ressources de votre fonction.

```
aws lambda add-permission --function-name my-function \
--statement-id iot-events --action "lambda:InvokeFunction" --principal
 iot.amazonaws.com
```

Vous devriez voir la sortie suivante:

```
{
    "Statement": "{\"Sid\":\"iot-events\",\"Effect\":\"Allow\",\"Principal\":
{\"Service\":\"iot.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction\",\"Resource\":
\"arn:aws:lambda:us-east-1:123456789012:function:my-function\"}"
}
```

Pour plus d'informations sur l'utilisation de Lambda avec AWS IoT, consultez Création d'une règle AWS Lambda.

Utilisation de Lambda pour traiter les enregistrements à partir d'Amazon Kinesis Data Streams

Vous pouvez utiliser une fonction Lambda pour traiter des enregistrements dans un flux de données Amazon Kinesis. Vous pouvez mapper une fonction Lambda à un consommateur à débit partagé (itérateur standard) Kinesis Data Streams ou à un consommateur à débit dédié avec diffusion améliorée. Pour les itérateurs standard, Lambda interroge chaque partition de votre flux Kinesis pour des enregistrements qui utilisent le protocole HTTP. Le mappage de source d'événement partage le débit de lecture avec d'autres utilisateurs de la partition.

Pour plus d'informations sur les flux de données Kinesis, consultez Lecture de données à partir d'Amazon Kinesis Data Streams.

> **Note**
>
> Kinesis facture chaque partition et, pour les diffusions améliorées, les données lues à partir du flux. Pour obtenir des informations de tarification, consultez Tarification Amazon Kinesis.

Flux d'interrogation et de mise en lots

Lambda lit les enregistrements du flux de données et invoque votre fonction de manière synchrone avec un événement contenant des enregistrements de flux. Lambda lit les registres par lots et invoque votre fonction pour les traiter. Chaque lot contient des enregistrements provenant d'un seul flux de données/partition.

Votre fonction Lambda est une application consommateur pour votre flux de données. Elle traite un lot d'enregistrements à la fois à partir de chaque partition. Vous pouvez mapper une fonction Lambda à un consommateur à débit partagé (itérateur standard) ou à un consommateur à débit dédié avec diffusion améliorée.

- Itérateur standard : Lambda interroge chaque partition de votre flux Kinesis afin d'obtenir des enregistrements à une fréquence de base d'une fois par seconde. Lorsque d'autres enregistrements sont disponibles, Lambda continue de traiter les lots jusqu'à ce que la fonction rattrape le flux. Le mappage de source d'événement partage le débit de lecture avec d'autres utilisateurs de la partition.
- Diffusion améliorée : pour réduire la latence et optimiser le débit en lecture, créez un consommateur de flux de données avec diffusion améliorée. Les consommateurs avec diffusion

améliorée obtiennent une connexion dédiée pour chaque partition qui n'a pas d'impact sur les autres applications lisant sur le flux. Les consommateurs de flux utilisent HTTP/2 afin de réduire la latence en transférant les enregistrements à Lambda via une connexion longue durée et en comprimant les en-têtes de requête. Vous pouvez créer un consommateur de flux avec l'API RegisterStreamConsumer de Kinesis.

```
aws kinesis register-stream-consumer \
--consumer-name con1 \
--stream-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream
```

Vous devriez voir la sortie suivante:

```
{
    "Consumer": {
        "ConsumerName": "con1",
        "ConsumerARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream/
consumer/con1:1540591608",
        "ConsumerStatus": "CREATING",
        "ConsumerCreationTimestamp": 1540591608.0
    }
}
```

Pour augmenter la vitesse à laquelle votre fonction traite les enregistrements, ajoutez des partitions à votre flux de données. Lambda traite les enregistrements de chaque partition dans l'ordre. Il arrête de traiter les enregistrements supplémentaires d'une partition si votre fonction renvoie une erreur. Plus de partitions signifient plus de lots traités en une seule fois, ce qui réduit l'impact des erreurs sur la simultanéité.

Si votre fonction ne peut pas augmenter sa capacité pour traiter le nombre total de lots simultanés, demandez une augmentation de quota ou réservez de la simultanéité pour votre fonction.

Par défaut, Lambda invoque votre fonction dès que des enregistrements sont disponibles. Si le lot que Lambda lit à partir de la source d'événements ne comprend qu'un seul enregistrement, Lambda envoie un seul registre à la fonction. Pour éviter d'invoquer la fonction avec un petit nombre de registres, vous pouvez indiquer à la source d'événement de les mettre en mémoire tampon pendant 5 minutes en configurant une fenêtre de traitement par lots. Avant d'invoquer la fonction, Lambda continue de lire les registres de la source d'événements jusqu'à ce qu'il ait rassemblé un lot complet, que la fenêtre de traitement par lot expire ou que le lot atteigne la limite de charge utile de 6 Mo. Pour de plus amples informations, consultez Comportement de traitement par lots.

> **Warning**
>
> Les mappages des sources d'événements Lambda traitent chaque événement au moins une fois, et le traitement des enregistrements peut être dupliqué. Pour éviter les problèmes potentiels liés à des événements dupliqués, nous vous recommandons vivement de rendre votre code de fonction idempotent. Pour en savoir plus, consultez Comment puis-je rendre ma fonction Lambda idempotente ? dans le Centre de connaissances AWS.

Lambda envoi le prochain lot pour traitement sans attendre que les extensions configurées soient terminées. En d'autres termes, vos extensions peuvent continuer à s'exécuter pendant que Lambda traite le prochain lot d'enregistrements. Cela peut causer des problèmes de limitation si vous enfreignez l'un des paramètres ou l'une des limites de simultanéité de votre compte. Pour détecter s'il s'agit d'un problème éventuel, surveillez vos fonctions et vérifiez si vous observez des métriques de simultanéité plus élevées que prévu pour votre mappage des sources d'événements. En raison de la brièveté des intervalles entre les invocations, Lambda peut brièvement signaler une utilisation simultanée supérieure au nombre de partitions. Cela peut être vrai même pour les fonctions Lambda sans extensions.

Configurez le paramètre ParallelizationFactor pour traiter une partition d'un flux de données Kinesis avec plusieurs invocations Lambda simultanément. Vous pouvez spécifier le nombre de lots simultanés que Lambda interroge à partir d'une partition via un facteur de parallélisation de 1 (par défaut) à 10. Par exemple, quand vous définissez `ParallelizationFactor` sur 2, vous pouvez avoir jusqu'à 200 invocations Lambda simultanés pour traiter 100 partitions de données Kinesis (bien que, dans la réalité, la métrique `ConcurrentExecutions` puisse indiquer une valeur différente). Cela permet d'augmenter le débit de traitement quand le volume de données est volatil et que la valeur du paramètre `IteratorAge` est élevée. Lorsque vous augmentez le nombre de lots simultanés par partition, Lambda assure toujours un traitement dans l'ordre au niveau clé de partition.

Vous pouvez également utiliser `ParallelizationFactor` avec l'agrégation Kinesis. Le comportement du mappage des sources d'événements varie selon que vous utilisez ou non une diffusion améliorée :

- Sans diffusion améliorée : tous les événements d'un événement agrégé doivent avoir la même clé de partition. La clé de partition doit également correspondre à celle de l'événement agrégé. Si les événements contenus dans l'événement agrégé ont des clés de partition différentes, Lambda ne peut garantir le traitement dans l'ordre des événements par clé de partition.

- Avec une diffusion améliorée : Lambda décode d'abord l'événement agrégé en événements individuels. L'événement agrégé peut avoir une clé de partition différente de celle des événements qu'il contient. Cependant, les événements qui ne correspondent pas à la clé de partition sont supprimés et perdus. Lambda ne traite pas ces événements et ne les envoie pas vers une destination en cas de panne configurée.

Exemple d'évènement

Example

```
{
    "Records": [
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692538361571095921575989136588898",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "approximateArrivalTimestamp": 1545084650.987
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
            "awsRegion": "us-east-2",
            "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-
stream"
        },
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692540925702759324208523137515618",
                "data": "VGhpcyBpcyBvbmx5IGEgdGVzdC4=",
                "approximateArrivalTimestamp": 1545084711.166
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
```

```
            "eventID":
 "shardId-000000000006:49590338271490256608559692540925702759324208523137515618",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
            "awsRegion": "us-east-2",
            "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-
stream"
        }
    ]
}
```

Traitement des enregistrements Amazon Kinesis Data Streams avec Lambda

Pour traiter les enregistrements Amazon Kinesis Data Streams avec Lambda, créez un mappage des sources d'événements Lambda. Vous pouvez mapper une fonction Lambda à un flux de données itérateur standard ou à un consommateur de diffusion améliorée. Pour de plus amples informations, veuillez consulter Flux d'interrogation et de mise en lots.

Créer un mappage des sources d'événements Kinesis

Pour invoquer votre fonction Lambda avec des enregistrements provenant de votre flux de données, créez un mappage des sources d'événements. Vous pouvez créer plusieurs mappages de source d'événement pour traiter les mêmes données avec plusieurs fonctions Lambda, ou pour traiter des éléments en provenance de plusieurs flux de données avec une seule fonction. Lorsque vous traitez des éléments à partir de plusieurs flux, chaque lot ne contient que des enregistrements provenant d'une seule partition ou d'un seul flux.

Vous pouvez configurer des mappages de sources d'événements pour traiter les enregistrements d'un flux dans un autre Compte AWS. Pour en savoir plus, veuillez consulter la section the section called "Mappages entre comptes".

Avant de créer un mappage des sources d'événements, vous devez autoriser votre fonction Lambda à lire à partir d'un flux de données Kinesis. Lambda a besoin des autorisations suivantes pour gérer les ressources liées à votre flux de données Kinesis :

- kinésie : DescribeStream
- kinésie : DescribeStreamSummary
- kinésie : GetRecords

- kinésie : GetShardIterator
- kinésie : ListShards
- kinésie : SubscribeToShard

La politique AWS gérée AWSLambdaKinesisExecutionRoleinclut ces autorisations. Ajoutez cette politique gérée à votre fonction comme décrit dans la procédure suivante.

> **Note**
>
> - Vous n'avez pas besoin de l'autorisation `kinesis:ListStreams` pour créer et gérer des mappages des sources d'événements pour Kinesis. Toutefois, si vous créez un mappage des sources d'événements dans la console et que vous ne disposez pas de cette autorisation, vous ne pourrez pas sélectionner un flux Kinesis dans une liste déroulante et la console affichera un message d'erreur. Pour créer le mappage des sources d'événements, vous devez saisir manuellement l'Amazon Resource Name (ARN) de votre flux.
> - Lambda effectue des appels d'API `kinesis:GetRecords` et `kinesis:GetShardIterator` lorsqu'il tente à nouveau des appels qui ont échoué.

AWS Management Console

Pour ajouter des autorisations Kinesis à votre fonction

1. Ouvrez la page Fonctions de la console Lambda et choisissez votre fonction.
2. Sous l'onglet Configuration, sélectionnez Autorisations.
3. Dans le volet Rôle d'exécution, sous Nom du rôle, choisissez le lien vers le rôle d'exécution de votre fonction. Ce lien ouvre la page de ce rôle dans la console IAM.
4. Dans le volet Politiques d'autorisations, choisissez Ajouter des autorisations, puis sélectionnez Attacher des politiques.
5. Dans le champ de recherche, entrez **`AWSLambdaKinesisExecutionRole`**.
6. Cochez la case en regard de la politique, puis choisissez Ajouter une autorisation.

AWS CLI

Pour ajouter des autorisations Kinesis à votre fonction

- Exécutez la commande de la CLI suivante pour ajouter la politique `AWSLambdaKinesisExecutionRole` au rôle d'exécution de votre fonction :

```
aws iam attach-role-policy \
--role-name MyFunctionRole \
--policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaKinesisExecutionRole
```

AWS SAM

Pour ajouter des autorisations Kinesis à votre fonction

- Dans la définition de votre fonction, ajoutez la propriété `Policies` comme indiqué dans l'exemple ci-dessous :

```
Resources:
  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./my-function/
      Handler: index.handler
      Runtime: nodejs24.x
      Policies:
        - AWSLambdaKinesisExecutionRole
```

Après avoir configuré les autorisations requises, créez le mappage des sources d'événements.

AWS Management Console

Pour créer le mappage des sources d'événements Kinesis

1. Ouvrez la page Fonctions de la console Lambda et choisissez votre fonction.
2. Dans le volet de Présentation de la fonction, choisissez Ajouter un déclencheur.
3. Sous Configuration du déclencheur, pour la source, sélectionnez Kinesis.
4. Sélectionnez le flux Kinesis pour lequel vous souhaitez créer le mappage des sources d'événements et, éventuellement, un consommateur de votre flux.

5. (Facultatif) Modifiez la Taille de lot, la Position de départ et la Fenêtre de traitement par lot de votre mappage des sources d'événements.
6. Choisissez Ajouter.

Lorsque vous créez le mappage de vos sources d'événements depuis la console, votre rôle IAM doit disposer des autorisations kinesis : ListStreams et kinesis :. ListStreamConsumers

AWS CLI

Pour créer le mappage des sources d'événements Kinesis

- Exécutez la commande de la CLI suivante pour créer un mappage des sources d'événements Kinesis. Choisissez votre propre taille de lot et votre position de départ en fonction de votre cas d'utilisation.

```
aws lambda create-event-source-mapping \
--function-name MyFunction \
--event-source-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream \
--starting-position LATEST \
--batch-size 100
```

Pour spécifier une fenêtre de traitement par lot, ajoutez l'option `--maximum-batching-window-in-seconds`. Pour plus d'informations sur l'utilisation de ce paramètre et d'autres paramètres, consultez create-event-source-mappingla référence des AWS CLI commandes.

AWS SAM

Pour créer le mappage des sources d'événements Kinesis

- Dans la définition de votre fonction, ajoutez la propriété `KinesisEvent` comme indiqué dans l'exemple ci-dessous :

```
Resources:
  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./my-function/
      Handler: index.handler
      Runtime: nodejs24.x
      Policies:
```

```
        - AWSLambdaKinesisExecutionRole
      Events:
        KinesisEvent:
          Type: Kinesis
          Properties:
            Stream: !GetAtt MyKinesisStream.Arn
            StartingPosition: LATEST
            BatchSize: 100

  MyKinesisStream:
    Type: AWS::Kinesis::Stream
    Properties:
      ShardCount: 1
```

Pour en savoir plus sur la création d'un mappage des sources d'événements pour Kinesis Data Streams AWS SAM in, consultez Kinesis dans AWS Serverless Application Model le manuel du développeur.

Position de départ du sondage et du stream

Sachez que l'interrogation des flux lors des mises à jour et de la création du mappage des sources d'événements est finalement cohérente.

- Lors de la création du mappage des sources d'événements, le démarrage de l'interrogation des événements depuis le flux peut prendre plusieurs minutes.
- Lors des mises à jour du mappage des sources d'événements, l'arrêt et le redémarrage de l'interrogation des événements depuis le flux peuvent prendre plusieurs minutes.

Ce comportement signifie que si vous spécifiez `LATEST` comme position de départ du flux, le mappage des sources d'événements peut manquer des événements lors de la création ou des mises à jour. Pour vous assurer de ne manquer aucun événement, spécifiez la position de départ du flux comme `TRIM_HORIZON` ou `AT_TIMESTAMP`.

Création d'un mappage des sources d'événements entre comptes

Amazon Kinesis Data Streams prend en charge les politiques basées sur les ressources. De ce fait, vous pouvez traiter les données ingérées dans un flux à l' Compte AWS aide d'une fonction Lambda dans un autre compte.

Pour créer un mappage de source d'événements pour votre fonction Lambda à l'aide d'un flux Kinesis dans un autre Compte AWS, vous devez configurer le flux à l'aide d'une politique basée sur les ressources afin d'autoriser votre fonction Lambda à lire des éléments. Pour savoir comment configurer votre stream afin d'autoriser l'accès entre comptes, consultez la section Partage de l'accès avec des AWS Lambda fonctions multicomptes dans le guide du développeur Amazon Kinesis Streams.

Une fois que vous avez configuré votre flux avec une politique basée sur les ressources qui donne à votre fonction Lambda les autorisations requises, créez le mappage des sources d'événements à l'aide de l'une des méthodes décrites dans la section précédente.

Si vous choisissez de créer votre mappage des sources d'événements à l'aide de la console Lambda, collez l'ARN de votre flux directement dans la zone de saisie. Si vous souhaitez spécifier un consommateur pour votre flux, le champ du flux est automatiquement rempli lorsque l'ARN du consommateur est collé.

Configuration d'une réponse par lots partielle avec Kinesis Data Streams et Lambda

Lors de l'utilisation et du traitement de données de streaming à partir d'une source d'événement, par défaut, Lambda n'effectue un point de contrôle sur le numéro de séquence le plus élevé d'un lot que lorsque celui-ci est un succès complet. Lambda traite tous les autres résultats comme un échec complet et recommence à traiter le lot jusqu'à atteindre la limite de nouvelles tentatives. Pour autoriser des succès partiels lors du traitement des lots à partir d'un flux, activez `ReportBatchItemFailures`. Autoriser des succès partiels peut permettre de réduire le nombre de nouvelles tentatives sur un enregistrement, mais cela n'empêche pas entièrement la possibilité de nouvelles tentatives dans un enregistrement réussi.

Pour activer `ReportBatchItemFailures`, incluez la valeur enum **`ReportBatchItemFailures`** dans la listeFunctionResponseTypes. Cette liste indique quels types de réponse sont activés pour votre fonction. Vous pouvez configurer cette liste lorsque vous créez ou mettez à jour un mappage des sources d'événements.

> **Note**
>
> Même lorsque le code de votre fonction renvoie des réponses d'échec partiel de lot, ces réponses ne seront pas traitées par Lambda à moins que la fonctionnalité

`ReportBatchItemFailures` soit explicitement activée pour votre mappage des sources d'événements.

Syntaxe du rapport

Lors de la configuration des rapports d'échec d'articles de lot, la classe `StreamsEventResponse` est renvoyée avec une liste d'échecs d'articles de lot. Vous pouvez utiliser un objet `StreamsEventResponse` pour renvoyer le numéro de séquence du premier enregistrement ayant échoué dans le lot. Vous pouvez également créer votre classe personnalisée en utilisant la syntaxe de réponse correcte. La structure JSON suivante montre la syntaxe de réponse requise :

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "<SequenceNumber>"
        }
    ]
}
```

Note

Si le tableau `batchItemFailures` contient plusieurs éléments, Lambda utilise l'enregistrement portant le numéro de séquence le plus bas comme point de contrôle. Lambda réessaie ensuite tous les enregistrements à partir de ce point de contrôle.

Conditions de réussite et d'échec

Lambda traite un lot comme un succès complet si vous renvoyez l'un des éléments suivants :

- Une liste `batchItemFailure` vide
- Une liste `batchItemFailure` nulle
- Une `EventResponse` vide
- Un nu `EventResponse`

Lambda traite un lot comme un échec complet si vous renvoyez l'un des éléments suivants :

- Une chaîne vid `itemIdentifier`
- Un `itemIdentifier` nul
- Un `itemIdentifier` avec un nom de clé incorrect

Lambda effectue des nouvelles tentatives en cas d'échec en fonction de votre stratégie de nouvelle tentative.

Diviser un lot

Si votre invocation échoue et que `BisectBatchOnFunctionError` est activé, le lot est divisé en deux quel que soit votre paramètre `ReportBatchItemFailures`.

Quand une réponse de succès partiel de lot est reçue et que les paramètres `BisectBatchOnFunctionError` et `ReportBatchItemFailures` sont activés, le lot est divisé au numéro de séquence renvoyé, et Lambda n'effectue de nouvelle tentative que sur les enregistrements restants.

Pour simplifier la mise en œuvre de la logique de réponse partielle par lots, pensez à utiliser l'utilitaire Batch Processor de Powertools for AWS Lambda, qui gère automatiquement ces complexités pour vous.

Voici quelques exemples de code de fonction qui renvoie la liste des messages ayant échoué IDs dans le lot :

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus sur GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots Kinesis avec Lambda à l'aide de .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using System.Text.Json.Serialization;
```

```
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegration;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task<StreamsEventResponse> FunctionHandler(KinesisEvent evnt,
 ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return new StreamsEventResponse();
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                return new StreamsEventResponse
                {
```

```
                    BatchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>
                    {
                        new StreamsEventResponse.BatchItemFailure
 { ItemIdentifier = record.Kinesis.SequenceNumber }
                    }
                };
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
        return new StreamsEventResponse();
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}

public class StreamsEventResponse
{
    [JsonPropertyName("batchItemFailures")]
    public IList<BatchItemFailure> BatchItemFailures { get; set; }
    public class BatchItemFailure
    {
        [JsonPropertyName("itemIdentifier")]
        public string ItemIdentifier { get; set; }
    }
}
```

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus sur GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots Kinesis avec Lambda à l'aide de Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, record := range kinesisEvent.Records {
  curRecordSequenceNumber := ""

  // Process your record
  if /* Your record processing condition here */ {
   curRecordSequenceNumber = record.Kinesis.SequenceNumber
  }

  // Add a condition to check if the record processing failed
  if curRecordSequenceNumber != "" {
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": curRecordSequenceNumber})
  }
 }

 kinesisBatchResponse := map[string]interface{}{
```

```
  "batchItemFailures": batchItemFailures,
 }
 return kinesisBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus sur GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots Kinesis avec Lambda à l'aide de Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessKinesisRecords implements RequestHandler<KinesisEvent,
 StreamsEventResponse> {

    @Override
    public StreamsEventResponse handleRequest(KinesisEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
```

```
        String curRecordSequenceNumber = "";

        for (KinesisEvent.KinesisEventRecord kinesisEventRecord :
 input.getRecords()) {
            try {
                //Process your record
                KinesisEvent.Record kinesisRecord =
 kinesisEventRecord.getKinesis();
                curRecordSequenceNumber = kinesisRecord.getSequenceNumber();

            } catch (Exception e) {
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse(batchItemFailures);
    }
}
```

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus sur GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots Kinesis avec Lambda à l'aide de JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
```

```
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

Signaler les défaillances d'éléments de lot Kinesis avec Lambda à l'aide de. TypeScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
  KinesisStreamBatchResponse,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
```

```
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<KinesisStreamBatchResponse> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  logger.info(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus sur GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots Kinesis avec Lambda à l'aide de PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $kinesisEvent = new KinesisEvent($event);
        $this->logger->info("Processing records");
        $records = $kinesisEvent->getRecords();
```

```
        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus sur GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots Kinesis avec Lambda à l'aide de Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["kinesis"]["sequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus sur GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots Kinesis avec Lambda à l'aide de Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  batch_item_failures = []

  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
```

```
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue StandardError => err
      puts "An error occurred #{err}"
      # Since we are working with streams, we can return the failed item
 immediately.
      # Lambda will immediately begin to retry processing from this failed item
 onwards.
      return { batchItemFailures: [{ itemIdentifier: record['kinesis']
['sequenceNumber'] }] }
    end
  end

  puts "Successfully processed #{event['Records'].length} records."
  { batchItemFailures: batch_item_failures }
end

def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('utf-8')
  # Placeholder for actual async work
  sleep(1)
  data
end
```

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus sur GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots Kinesis avec Lambda à l'aide de Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::kinesis::KinesisEvent,
    kinesis::KinesisEventRecord,
    streams::{KinesisBatchItemFailure, KinesisEventResponse},
```

```
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) ->
 Result<KinesisEventResponse, Error> {
    let mut response = KinesisEventResponse {
        batch_item_failures: vec![],
    };

    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in &event.payload.records {
        tracing::info!(
            "EventId: {}",
            record.event_id.as_deref().unwrap_or_default()
        );

        let record_processing_result = process_record(record);

        if record_processing_result.is_err() {
            response.batch_item_failures.push(KinesisBatchItemFailure {
                item_identifier: record.kinesis.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(response)
}

fn process_record(record: &KinesisEventRecord) -> Result<(), Error> {
    let record_data = std::str::from_utf8(record.kinesis.data.as_slice());
```

```
    if let Some(err) = record_data.err() {
        tracing::error!("Error: {}", err);
        return Err(Error::from(err));
    }

    let record_data = record_data.unwrap_or_default();

    // do something interesting with the data
    tracing::info!("Data: {}", record_data);

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

Utilisation de Powertools pour le AWS Lambda traitement par lots

L'utilitaire de traitement par lots de Powertools for gère AWS Lambda automatiquement la logique de réponse partielle par lots, réduisant ainsi la complexité de la mise en œuvre du signalement des défaillances par lots. Voici des exemples d'utilisation de l'utilitaire de traitement par lots :

Python

> **Note**
>
> Pour des exemples complets et des instructions de configuration, consultez la documentation de l'utilitaire de traitement par lots.

Traitement des enregistrements de flux Kinesis Data Streams à l'aide d' AWS Lambda un processeur par lots.

```
import json
from aws_lambda_powertools import Logger
from aws_lambda_powertools.utilities.batch import BatchProcessor, EventType,
 process_partial_response
from aws_lambda_powertools.utilities.data_classes import KinesisEvent
from aws_lambda_powertools.utilities.typing import LambdaContext

processor = BatchProcessor(event_type=EventType.KinesisDataStreams)
logger = Logger()

def record_handler(record):
    logger.info(record)
    # Your business logic here
    # Raise an exception to mark this record as failed

def lambda_handler(event, context: LambdaContext):
    return process_partial_response(
        event=event,
        record_handler=record_handler,
        processor=processor,
        context=context
    )
```

TypeScript

> **Note**
>
> Pour des exemples complets et des instructions de configuration, consultez la documentation de l'utilitaire de traitement par lots.

Traitement des enregistrements de flux Kinesis Data Streams à l'aide d' AWS Lambda un processeur par lots.

```
import { BatchProcessor, EventType, processPartialResponse } from '@aws-lambda-
powertools/batch';
import { Logger } from '@aws-lambda-powertools/logger';
import type { KinesisEvent, Context } from 'aws-lambda';
```

```
const processor = new BatchProcessor(EventType.KinesisDataStreams);
const logger = new Logger();

const recordHandler = async (record: any): Promise<void> => {
    logger.info('Processing record', { record });
    // Your business logic here
    // Throw an error to mark this record as failed
};

export const handler = async (event: KinesisEvent, context: Context) => {
    return processPartialResponse(event, recordHandler, processor, {
        context,
    });
};
```

Java

> **Note**
>
> Pour des exemples complets et des instructions de configuration, consultez la documentation de l'utilitaire de traitement par lots.

Traitement des enregistrements de flux Kinesis Data Streams à l'aide d' AWS Lambda un processeur par lots.

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import software.amazon.lambda.powertools.batch.BatchMessageHandlerBuilder;
import software.amazon.lambda.powertools.batch.handler.BatchMessageHandler;

public class KinesisStreamBatchHandler implements RequestHandler<KinesisEvent,
 StreamsEventResponse> {

    private final BatchMessageHandler<KinesisEvent, StreamsEventResponse> handler;

    public KinesisStreamBatchHandler() {
        handler = new BatchMessageHandlerBuilder()
                .withKinesisBatchHandler()
                .buildWithRawMessageHandler(this::processMessage);
```

```
    }

    @Override
    public StreamsEventResponse handleRequest(KinesisEvent kinesisEvent, Context
 context) {
        return handler.processBatch(kinesisEvent, context);
    }

    private void processMessage(KinesisEvent.KinesisEventRecord kinesisEventRecord,
 Context context) {
        // Process the stream record
    }
}
```

.NET

> **Note**
>
> Pour des exemples complets et des instructions de configuration, consultez la documentation de l'utilitaire de traitement par lots.

Traitement des enregistrements de flux Kinesis Data Streams à l'aide d' AWS Lambda un processeur par lots.

```
using System;
using System.Threading;
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using Amazon.Lambda.Serialization.SystemTextJson;
using AWS.Lambda.Powertools.BatchProcessing;

[assembly: LambdaSerializer(typeof(DefaultLambdaJsonSerializer))]

namespace HelloWorld;

public class OrderEvent
{
    public string? OrderId { get; set; }
    public string? CustomerId { get; set; }
    public decimal Amount { get; set; }
```

```
    public DateTime OrderDate { get; set; }
}

internal class TypedKinesisRecordHandler : ITypedRecordHandler<OrderEvent>
{
    public async Task<RecordHandlerResult> HandleAsync(OrderEvent orderEvent,
 CancellationToken cancellationToken)
    {
        if (string.IsNullOrEmpty(orderEvent.OrderId))
        {
            throw new ArgumentException("Order ID is required");
        }

        return await Task.FromResult(RecordHandlerResult.None);
    }
}

public class Function
{
    [BatchProcessor(TypedRecordHandler = typeof(TypedKinesisRecordHandler))]
    public BatchItemFailuresResponse HandlerUsingTypedAttribute(KinesisEvent _)
    {
        return TypedKinesisStreamBatchProcessor.Result.BatchItemFailuresResponse;
    }
}
```

Retenir les enregistrements de lots supprimés pour une source d’événements de flux de données Kinesis dans Lambda

La gestion des erreurs pour les mappages des sources d’événements Kinesis n’est pas la même selon si l’erreur se produit avant que la fonction ne soit invoquée ou pendant l’invocation de la fonction :

- Avant l'appel : si un mappage de source d'événement Lambda ne parvient pas à appeler la fonction en raison d'un ralentissement ou d'autres problèmes, il réessaie jusqu'à ce que les enregistrements expirent ou dépassent l'âge maximum configuré sur le mappage de source d'événement (). MaximumRecordAgeInSeconds
- Pendant l'appel : si la fonction est invoquée mais renvoie une erreur, Lambda réessaie jusqu'à ce que les enregistrements expirent, dépassent l'âge maximum MaximumRecordAgeInSeconds() ou atteignent le quota de nouvelles tentatives configuré (). MaximumRetryAttempts Pour les erreurs de

fonctionnement, vous pouvez également configurer BisectBatchOnFunctionError, ce qui divise un lot défaillant en deux lots plus petits, isolant ainsi les mauvais enregistrements et évitant les délais d'attente. Le fractionnement des lots ne consomme pas le quota de nouvelles tentatives.

Si les mesures de gestion des erreurs échouent, Lambda ignore les enregistrements et poursuit le traitement des lots du flux. Avec les paramètres par défaut, cela signifie qu'un enregistrement défectueux peut bloquer le traitement sur la partition affectée pendant jusqu'à une semaine. Pour éviter cela, configurez le mappage de source d'événement de votre fonction avec un nombre raisonnable de nouvelles tentatives et un âge maximum d'enregistrement correspondant à votre cas d'utilisation.

Configuration des destinations pour les invocations ayant échoué

Pour retenir les enregistrements des invocations de mappage de sources d'événements qui ont échoué, ajoutez une destination au mappage des sources d'événements de votre fonction. Chaque enregistrement envoyé à la destination est un document JSON contenant les métadonnées sur l'invocation ayant échoué. Pour les destinations Amazon S3, Lambda envoie également l'intégralité de l'enregistrement d'invocation avec les métadonnées. Vous pouvez configurer n'importe quelle rubrique Amazon SNS, n'importe quelle file d'attente Amazon SQS, n'importe quel compartiment Amazon S3 ou Kafka comme destination.

Avec les destinations Amazon S3, vous pouvez utiliser la fonctionnalité Notifications d'événements Amazon S3 pour recevoir des notifications lorsque des objets sont chargés dans votre compartiment S3 de destination. Vous pouvez également configurer les notifications d'événements S3 pour invoquer une autre fonction Lambda afin d'effectuer un traitement automatique des lots ayant échoué.

Votre rôle d'exécution doit disposer d'autorisations pour la destination :

- Pour une destination SQS : sqs : SendMessage
- Pour une destination SNS : SNS:Publish
- Pour une destination S3 : s3 : PutObject et s3 : ListBucket
- Pour une destination Kafka : kafka-cluster : WriteData

Vous pouvez configurer un sujet Kafka comme destination en cas d'échec pour vos mappages de sources d'événements Kafka. Lorsque Lambda ne parvient pas à traiter les enregistrements après avoir épuisé toutes les tentatives ou lorsque les enregistrements dépassent l'âge maximum, Lambda

envoie les enregistrements ayant échoué à la rubrique Kafka spécifiée pour un traitement ultérieur. Reportez-vous à the section called “Destination de Kafka en cas d'échec”.

Si vous avez activé le chiffrement avec votre propre clé KMS pour une destination S3, le rôle d'exécution de votre fonction doit également être autorisé à appeler kms : GenerateDataKey. Si la clé KMS et la destination du compartiment S3 se trouvent dans un compte différent de celui de votre fonction Lambda et de votre rôle d'exécution, configurez la clé KMS pour qu'elle approuve le rôle d'exécution à autoriser. kms: GenerateDataKey

Pour configurer une destination en cas de panne à l’aide de la console, procédez comme suit :

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Sous Function overview (Vue d’ensemble de la fonction), choisissez Add destination (Ajouter une destination).
4. Pour Source, choisissez Invocation du mappage des sources d’événements.
5. Pour le mappage des sources d’événements, choisissez une source d’événements configurée pour cette fonction.
6. Pour Condition, sélectionnez En cas d’échec. Pour les invocations de mappage des sources d’événements, il s’agit de la seule condition acceptée.
7. Pour Type de destination, choisissez le type de destination auquel Lambda envoie les enregistrements d’invocation.
8. Pour Destination, choisissez une ressource.
9. Choisissez Enregistrer.

Vous pouvez également configurer une destination en cas de panne à l'aide de AWS Command Line Interface (AWS CLI). Par exemple, la create-event-source-mappingcommande suivante ajoute un mappage de source d'événement avec une destination SQS en cas de défaillance pour : `MyFunction`

```
aws lambda create-event-source-mapping \
--function-name "MyFunction" \
--event-source-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-
east-1:123456789012:dest-queue"}}'
```

La update-event-source-mappingcommande suivante met à jour le mappage d'une source d'événements afin d'envoyer les enregistrements d'invocation ayant échoué vers une destination SNS après deux tentatives, ou si les enregistrements datent de plus d'une heure.

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--maximum-retry-attempts 2 \
--maximum-record-age-in-seconds 3600 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sns:us-
east-1:123456789012:dest-topic"}}'
```

Les paramètres mis à jour sont appliqués de façon asynchrone et ne sont pas reflétés dans la sortie tant que le processus n'est pas terminé. Utilisez la commande get-event-source-mapping pour afficher l'état actuel.

Pour supprimer une destination, entrez une chaîne vide comme argument du paramètre `destination-config` :

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

Pratiques exemplaires en matière de sécurité pour les destinations Amazon S3

La suppression d'un compartiment S3 configuré comme destination sans supprimer la destination de la configuration de votre fonction peut engendrer un risque de sécurité. Si un autre utilisateur connaît le nom de votre compartiment de destination, il peut recréer le compartiment dans son Compte AWS. Les enregistrements des invocations ayant échoué seront envoyés dans son compartiment, exposant potentiellement les données de votre fonction.

> **Warning**
>
> Pour vous assurer que les enregistrements d'invocation de votre fonction ne peuvent pas être envoyés vers un compartiment S3 d'un autre Compte AWS, ajoutez une condition au rôle d'exécution de votre fonction qui limite `s3:PutObject` les autorisations aux compartiments de votre compte.

L'exemple suivant présente une politique IAM qui limite les autorisations `s3:PutObject` de votre fonction aux seuls compartiments de votre compte. Cette politique donne également à Lambda l'autorisation `s3:ListBucket` dont il a besoin pour utiliser un compartiment S3 comme destination.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "S3BucketResourceAccountWrite",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::*/*",
                "arn:aws:s3:::*"
            ],
            "Condition": {
                "StringEquals": {
                    "s3:ResourceAccount": "111122223333"
                }
            }
        }
    ]
}
```

Pour ajouter une politique d'autorisations au rôle d'exécution de votre fonction à l'aide du AWS Management Console ou AWS CLI, reportez-vous aux instructions des procédures suivantes :

Console

Pour ajouter une politique d'autorisations au rôle d'exécution d'une fonction (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction Lambda dont vous voulez modifier le rôle d'exécution.
3. Sous l'onglet Configuration, sélectionnez Autorisations.
4. Sous l'onglet Rôle d'exécution, sélectionnez le Nom du rôle de votre fonction pour ouvrir la page de console IAM du rôle.
5. Ajoutez une politique d'autorisations de au rôle en procédant comme suit :

a. Dans le volet Politiques d'autorisations, choisissez Ajouter des autorisations, puis Créer une politique en ligne.

b. Dans l'Éditeur de politique, sélectionnez JSON.

c. Collez la politique que vous souhaitez ajouter dans l'éditeur (en remplacement du JSON existant), puis choisissez Suivant.

d. Sous Détails de la politique, saisissez un Nom de la politique.

e. Choisissez Create Policy (Créer une politique).

AWS CLI

Pour ajouter une politique d'autorisations au rôle d'exécution d'une fonction (CLI)

1. Créez un document de politique JSON avec les autorisations requises et enregistrez-le dans un répertoire local.
2. Utilisez la commande `put-role-policy` de la CLI IAM pour ajouter des autorisations au rôle d'exécution de votre fonction. Exécutez la commande suivante depuis le répertoire dans lequel vous avez enregistré votre document de politique JSON et remplacez le nom du rôle, le nom de la politique et le document de politique par vos propres valeurs.

```
aws iam put-role-policy \
--role-name my_lambda_role \
--policy-name LambdaS3DestinationPolicy \
--policy-document file://my_policy.json
```

Exemple d'enregistrement d'invocation Amazon SNS et Amazon SQS

L'exemple suivant montre ce que Lambda envoie à une file d'attente SQS ou à une rubrique SNS en cas d'échec de l'invocation d'une source d'événement Kinesis. Étant donné que Lambda envoie uniquement les métadonnées pour ces types de destination, utilisez les champs `streamArn`, `shardId`, `startSequenceNumber` et `endSequenceNumber` pour obtenir l'enregistrement original complet. Tous les champs présentés dans la propriété `KinesisBatchInfo` seront toujours présents.

```
{
    "requestContext": {
        "requestId": "c9b8fa9f-5a7f-xmpl-af9c-0c604cde93a5",
```

```
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KinesisBatchInfo": {
        "shardId": "shardId-000000000001",
        "startSequenceNumber":
 "49601189658422359378836298521827638475320189012309704722",
        "endSequenceNumber":
 "49601189658422359378836298522902373528957594348623495186",
        "approximateArrivalOfFirstRecord": "2019-11-14T00:38:04.835Z",
        "approximateArrivalOfLastRecord": "2019-11-14T00:38:05.580Z",
        "batchSize": 500,
        "streamArn": "arn:aws:kinesis:us-east-2:123456789012:stream/mystream"
    }
}
```

Vous pouvez utiliser ces informations pour récupérer les enregistrements concernés à partir du flux à des fins de résolution de problèmes. Les enregistrements réels n'étant pas inclus, vous devez les récupérer du flux avant qu'ils expirent et soient perdus.

Exemple d'enregistrement d'invocation Amazon S3

L'exemple suivant montre ce que Lambda envoie à un compartiment Amazon S3 ou à une invocation de source d'événement Kinesis. Outre tous les champs de l'exemple précédent pour les destinations SQS et SNS, le champ `payload` contient l'enregistrement d'invocation d'origine sous forme de chaîne JSON échappée.

```
{
    "requestContext": {
        "requestId": "c9b8fa9f-5a7f-xmpl-af9c-0c604cde93a5",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
```

```
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KinesisBatchInfo": {
        "shardId": "shardId-000000000001",
        "startSequenceNumber":
 "49601189658422359378836298521827638475320189012309704722",
        "endSequenceNumber":
 "49601189658422359378836298522902373528957594348623495186",
        "approximateArrivalOfFirstRecord": "2019-11-14T00:38:04.835Z",
        "approximateArrivalOfLastRecord": "2019-11-14T00:38:05.580Z",
        "batchSize": 500,
        "streamArn": "arn:aws:kinesis:us-east-2:123456789012:stream/mystream"
    },
    "payload": "<Whole Event>" // Only available in S3
}
```

L'objet S3 contenant l'enregistrement d'invocation utilise la convention de dénomination suivante :

```
aws/lambda/<ESM-UUID>/<shardID>/YYYY/MM/DD/YYYY-MM-DDTHH.MM.SS-<Random UUID>
```

Implémentation du traitement des flux de données Kinesis avec état dans Lambda

Les fonctions Lambda peuvent exécuter des applications de traitement de flux continu. Un flux représente des données illimitées qui circulent en continu dans votre application. Pour analyser les informations provenant de cette entrée de mise à jour continue, vous pouvez lier les enregistrements inclus à l'aide d'une fenêtre de temps définie.

Les fenêtres bascules sont des fenêtres temporelles distinctes qui s'ouvrent et se ferment à intervalles réguliers. Par défaut, les invocations Lambda sont sans état : vous ne pouvez pas les utiliser pour traiter des données sur plusieurs invocations continues sans base de données externe. Cependant, avec les fenêtres bascules, vous pouvez maintenir votre état au long des invocations. Cet état contient le résultat global des messages précédemment traités pour la fenêtre actuelle. Votre état peut être d'un maximum de 1 Mo par partition. S'il dépasse cette taille, Lambda met fin précocement à la fenêtre de traitement.

Chaque enregistrement d'un flux appartient à une fenêtre spécifique. La fonction Lambda traitera chaque enregistrement au moins une fois. Toutefois, elle ne garantit pas un seul traitement pour chaque enregistrement. Dans de rares cas, tels que pour la gestion des erreurs, certains enregistrements peuvent être sujet à de multiples traitements. Les dossiers sont toujours traités dans l'ordre dès la première fois. Si les enregistrements sont traités plusieurs fois, ils peuvent être traités dans le désordre.

Regroupement et traitement

Votre fonction gérée par l'utilisateur est invoquée tant pour l'agrégation que pour le traitement des résultats finaux de celle-ci. Lambda regroupe tous les enregistrements reçus dans la fenêtre. Vous pouvez recevoir ces enregistrements en plusieurs lots, chacun sous forme d'invocation séparée. Chaque invocation reçoit un état. Ainsi, lorsque vous utilisez des fenêtres bascules, votre réponse de fonction Lambda doit contenir une propriété `state`. Si la réponse ne contient pas de propriété `state`, Lambda considère qu'il s'agit d'une invocation ayant échoué. Pour satisfaire à cette condition, votre fonction peut renvoyer un objet `TimeWindowEventResponse` ayant la forme JSON suivante :

Example **`TimeWindowEventResponse`** values

```
{
    "state": {
        "1": 282,
        "2": 715
    },
    "batchItemFailures": []
}
```

> **Note**
>
> Pour les fonctions Java, nous vous recommandons d'utiliser `Map<String, String>` pour représenter l'état.

À la fin de la fenêtre, l'indicateur `isFinalInvokeForWindow` est défini sur `true` pour indiquer qu'il s'agit de l'état final et qu'il est prêt pour le traitement. Après le traitement, la fenêtre et votre invocation final se terminent, puis l'état est supprimé.

À la fin de votre fenêtre, Lambda applique un traitement final pour les actions sur les résultats de l'agrégation. Votre traitement final est invoqué de manière synchrone. Une fois l'invocation réussie,

votre fonction contrôle le numéro de séquence et le traitement du flux continue. Si l'invocation échoue, votre fonction Lambda suspend le traitement ultérieur jusqu'à ce que l'invocation soit réussie.

Example KinesisTimeWindowEvent

```
{
    "Records": [
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692538361571095921575989136588898",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "approximateArrivalTimestamp": 1607497475.000
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-kinesis-role",
            "awsRegion": "us-east-1",
            "eventSourceARN": "arn:aws:kinesis:us-east-1:123456789012:stream/lambda-
stream"
        }
    ],
    "window": {
        "start": "2020-12-09T07:04:00Z",
        "end": "2020-12-09T07:06:00Z"
    },
    "state": {
        "1": 282,
        "2": 715
    },
    "shardId": "shardId-000000000006",
    "eventSourceARN": "arn:aws:kinesis:us-east-1:123456789012:stream/lambda-stream",
    "isFinalInvokeForWindow": false,
    "isWindowTerminatedEarly": false
}
```

Configuration

Vous pouvez configurer des fenêtres bascule lorsque vous créez ou mettez à jour un mappage de source d'événement. Pour configurer une fenêtre bascule, précisez la fenêtre en secondes (TumblingWindowInSeconds). L'exemple de commande AWS Command Line Interface (AWS CLI) suivant crée un mappage de source d'événement de streaming dont la fenêtre bascule est de 120 secondes. La fonction Lambda définie pour l'agrégation et le traitement est nommée `tumbling-window-example-function`.

```
aws lambda create-event-source-mapping \
--event-source-arn arn:aws:kinesis:us-east-1:123456789012:stream/lambda-stream \
--function-name tumbling-window-example-function \
--starting-position TRIM_HORIZON \
--tumbling-window-in-seconds 120
```

Lambda détermine les limites des fenêtres bascule en fonction de l'heure à laquelle les enregistrements ont été insérés dans le flux. Tous les enregistrements ont un horodatage approximatif disponible que Lambda utilise pour déterminer des limites.

Les agrégations de fenêtres bascule ne prennent pas en charge le repartitionnement. Quand une partition prend fin, Lambda considère que la fenêtre de traitement actuelle est fermée, et les partitions enfants entament leur propre fenêtre de traitement dans un nouvel état. Lorsqu'aucun nouvel enregistrement n'est ajouté à la fenêtre actuelle, Lambda attend jusqu'à 2 minutes avant de supposer que la fenêtre est terminée. Cela permet de garantir que la fonction lit tous les enregistrements de la fenêtre actuelle, même si les enregistrements sont ajoutés par intermittence.

Les fenêtres bascule prennent complètement en charge les stratégies de nouvelle tentative existantes `maxRetryAttempts` et `maxRecordAge`.

Example Handler.py – Agrégation et traitement

La fonction Python suivante montre comment regrouper et traiter votre état final :

```
def lambda_handler(event, context):
    print('Incoming event: ', event)
    print('Incoming state: ', event['state'])

#Check if this is the end of the window to either aggregate or process.
    if event['isFinalInvokeForWindow']:
        # logic to handle final state of the window
```

```
        print('Destination invoke')
    else:
        print('Aggregate invoke')

#Check for early terminations
    if event['isWindowTerminatedEarly']:
        print('Window terminated early')

    #Aggregation logic
    state = event['state']
    for record in event['Records']:
        state[record['kinesis']['partitionKey']] = state.get(record['kinesis']
['partitionKey'], 0) + 1

    print('Returning state: ', state)
    return {'state': state}
```

Paramètres Lambda pour les mappages des sources d'événements Amazon Kinesis Data Streams

Tous les mappages de sources d'événements Lambda partagent les mêmes opérations d'API CreateEventSourceMapping et UpdateEventSourceMapping. Cependant, seuls certains paramètres s'appliquent à Kinesis.

Paramètre	Obligatoire	Par défaut	Remarques
BatchSize	N	100	Maximum : 10 000.
BisectBatchOnFunctionError	N	FAUX	none
DestinationConfig	N	N/A	File d'attente Amazon SQS ou destination de rubrique Amazon SNS pour les enregistrements supprimés. Pour de plus amples informations, consultez Configuration des

Paramètre	Obligatoire	Par défaut	Remarques
			destinations pour les invocations ayant échoué.
Activé	N	VRAI	none
EventSourceArn	Y	N/A	ARN du flux de données ou d'un consommateur de flux
FunctionName	Y	N/A	none
FunctionResponseTypes	N	N/A	Pour permettre à votre fonction de signaler des échecs spécifiques dans un lot, incluez la valeur `ReportBatchItemFailures` dans `FunctionResponseTypes` . Pour de plus amples informations, consultez Configuration d'une réponse par lots partielle avec Kinesis Data Streams et Lambda.
MaximumBatchingWindowInSeconds	N	0	none

Paramètre	Obligatoire	Par défaut	Remarques
MaximumRecordAgeInSeconds	N	-1	-1 signifie infini : Lambda ne supprime pas les enregistrements (les paramètres de conservation des données Kinesis Data Streams s'appliquent toujours) Minimum : -1 Maximum : 604 800
MaximumRetryAttempts	N	-1	-1 signifie infini : les registres qui ont échoué sont réessayés jusqu'à ce que le registre expire. Minimum : -1 Maximum : 10 000.
ParallelizationFactor	N	1	Maximum : 10
StartingPosition	O	N/A	AT_TIMESTAMP, TRIM_HORIZON ou DERNIER

Paramètre	Obligatoire	Par défaut	Remarques
StartingPositionTimestamp	N	N/A	Valide uniquement si StartingPosition est défini sur AT_TIMESTAMP. L'heure à partir de laquelle commencer la lecture, en secondes au format horaire Unix.
TumblingWindowInSeconds	N	N/A	Minimum : 0 Maximum : 900

Utilisation du filtrage des événements avec une source d'événement Kinesis

Vous pouvez utiliser le filtrage d'événements pour contrôler les enregistrements d'un flux ou d'une file d'attente que Lambda envoie à votre fonction. Pour obtenir des informations générales sur le fonctionnement du filtrage des événements, consultez the section called "Filtrage des événements".

Cette section porte sur le filtrage des événements pour les sources Kinesis.

> **Note**
>
> Les mappages des sources d'événements Kinesis prennent uniquement en charge le filtrage sur la clé `data`.

Rubriques

- Notions de base du filtrage des événements Kinesis
- Filtrage des enregistrements agrégés de Kinesis

Notions de base du filtrage des événements Kinesis

Supposons qu'un producteur insère des données au format JSON dans votre flux de données Kinesis. Un exemple d'enregistrement ressemblerait à ce qui suit, avec les données JSON converties en chaîne encodée en Base64 dans le champ `data`.

```
{
    "kinesis": {
        "kinesisSchemaVersion": "1.0",
        "partitionKey": "1",
        "sequenceNumber": "49590338271490256608559692538361571095921575989136588898",
        "data":
 "eyJSZWNvcmROdW1iZXIiOiAiMDAwMSIsICJUaW1lU3RhbXAiOiAieXl5eS1tbS1kZFRoaDptbTpzcyIsICJSZXF1ZXN0Q
        "approximateArrivalTimestamp": 1545084650.987
        },
    "eventSource": "aws:kinesis",
    "eventVersion": "1.0",
    "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
    "eventName": "aws:kinesis:record",
    "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
    "awsRegion": "us-east-2",
    "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream"
}
```

Tant que les données insérées dans le flux par le producteur sont des données JSON valides, vous pouvez utiliser le filtrage d'événements pour filtrer les enregistrements à l'aide de la clé `data`. Supposons qu'un producteur insère des enregistrements dans votre flux Kinesis au format JSON suivant.

```
{
    "record": 12345,
    "order": {
        "type": "buy",
        "stock": "ANYCO",
        "quantity": 1000
        }
}
```

Pour filtrer uniquement les enregistrements dont le type d'ordre est « buy », l'objet `FilterCriteria` serait le suivant.

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : { \"order\" : { \"type\" : [ \"buy\" ] } } }"
        }
    ]
}
```

Pour plus de clarté, voici la valeur du `Pattern` de filtre étendu en JSON simple :

```
{
    "data": {
        "order": {
            "type": [ "buy" ]
            }
      }
}
```

Vous pouvez ajouter votre filtre à l'aide de la console, d'AWS CLI ou d'un modèle AWS SAM.

Console

Pour ajouter ce filtre à l'aide de la console, suivez les instructions de Attacher des critères de filtre à un mappage de sources d'événements (console) et saisissez la chaîne suivante comme critère de filtre.

```
{ "data" : { "order" : { "type" : [ "buy" ] } } }
```

AWS CLI

Pour créer un nouveau mappage des sources d'événements avec ces critères de filtrage à l'aide de l'outil AWS Command Line Interface (AWS CLI), exécutez la commande suivante.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:kinesis:us-east-2:123456789012:stream/my-stream \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"order\" : { \"type
\" : [ \"buy\" ] } } }"}]}'
```

Pour ajouter ces critères de filtre à un mappage des sources d'événements existant, exécutez la commande suivante.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"order\" : { \"type
\" : [ \"buy\" ] } } }"}]}'
```

AWS SAM

Pour ajouter ce filtre à l'aide d'AWS SAM, ajoutez l'extrait suivant au modèle YAML de votre source d'événement.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "data" : { "order" : { "type" : [ "buy" ] } } }'
```

Pour filtrer correctement les événements provenant de sources Kinesis, le champ de données et vos critères de filtre pour le champ de données doivent être au format JSON valide. Si l'un ou l'autre des champs n'est pas dans un format JSON valide, Lambda rejette le message ou lance une exception. Le tableau suivant résume le comportement spécifique :

Format des données entrantes	Pas de modèle de filtre pour les propriétés des données	Action obtenue.
JSON valide	JSON valide	Lambda filtre en fonction de vos critères de filtre.
JSON valide	Pas de modèle de filtre pour les propriétés des données	Lambda filtre (uniquement selon les autres propriétés de métadonnées) en fonction de vos critères de filtre.
JSON valide	Non JSON	Lambda lance une exception au moment de la création ou de la mise à jour du mappage de sources d'événements. Le modèle de filtre des propriétés de données doit être au format JSON valide.

Format des données entrantes	Pas de modèle de filtre pour les propriétés des données	Action obtenue.
Non JSON	JSON valide	Lambda rejette l'enregis trement.
Non JSON	Pas de modèle de filtre pour les propriétés des données	Lambda filtre (uniquement selon les autres propriétés de métadonnées) en fonction de vos critères de filtre.
Non JSON	Non JSON	Lambda lance une exception au moment de la création ou de la mise à jour du mappage de sources d'événements. Le modèle de filtre des propriétés de données doit être au format JSON valide.

Filtrage des enregistrements agrégés de Kinesis

Avec Kinesis, vous pouvez agréger plusieurs enregistrements en un seul enregistrement Kinesis Data Streams pour augmenter le débit de vos données. Lambda ne peut appliquer des critères de filtrage aux enregistrements agrégés que lorsque vous utilisez la distribution ramifiée améliorée de Kinesis. Le filtrage des enregistrements agrégés avec Kinesis standard n'est pas pris en charge. Lorsque vous utilisez la distribution ramifiée améliorée, vous configurez un consommateur Kinesis à débit dédié pour qu'il serve de déclencheur à votre fonction Lambda. Lambda filtre alors les enregistrements agrégés et ne transmet que les enregistrements qui répondent à vos critères de filtrage.

Pour en savoir plus sur l'agrégation d'enregistrements Kinesis, reportez-vous à la section Agrégation sur la page Concepts clés de Kinesis Producer Library (KPL). Pour en savoir plus sur l'utilisation de Lambda avec la distribution ramifiée améliorée de Kinesis, consultez Increasing real-time stream processing performance with Amazon Kinesis Data Streams enhanced fan-out and AWS Lambda sur le blog AWS compute.

Tutoriel : Utilisation de Lambda avec les flux de données Kinesis

Dans ce tutoriel, vous créez une fonction Lambda pour consommer des événements à partir d'un flux de données Amazon Kinesis.

1. L'application personnalisée écrit les enregistrements dans le flux.
2. AWS Lambda interroge le flux et, lorsqu'il détecte de nouveaux enregistrements dans le flux, invoque votre fonction Lambda.
3. AWS Lambda exécute la fonction Lambda en assumant le rôle d'exécution que vous avez spécifié au moment de la création de la fonction Lambda.

Conditions préalables

Installez le AWS Command Line Interface

Si vous ne l'avez pas encore installé AWS Command Line Interface, suivez les étapes décrites dans la section Installation ou mise à jour de la dernière version du AWS CLI pour l'installer.

Ce tutoriel nécessite un terminal de ligne de commande ou un shell pour exécuter les commandes. Sous Linux et macOS, utilisez votre gestionnaire de shell et de package préféré.

> **Note**
>
> Sous Windows, certaines commandes CLI Bash que vous utilisez couramment avec Lambda (par exemple `zip`) ne sont pas prises en charge par les terminaux intégrés du système d'exploitation. Installez le sous-système Windows pour Linux afin d'obtenir une version intégrée à Windows d'Ubuntu et Bash.

Créer le rôle d'exécution

Créez le rôle d'exécution qui autorise votre fonction à accéder aux AWS ressources.

Pour créer un rôle d'exécution

1. Ouvrez la page Roles (Rôles) dans la console IAM.
2. Sélectionnez Créer un rôle.
3. Créez un rôle avec les propriétés suivantes :

- Entité de confiance – AWS Lambda.
- Autorisations — AWSLambdaKinesisExecutionRole.
- Nom de rôle – **`lambda-kinesis-role`**.

La AWSLambdaKinesisExecutionRolepolitique dispose des autorisations dont la fonction a besoin pour lire des éléments provenant de Kinesis et écrire des journaux dans Logs. CloudWatch

Créer la fonction

Créez une fonction Lambda qui traite vos messages Kinesis. Le code de fonction enregistre l'ID d'événement et les données d'événement de l'enregistrement Kinesis dans Logs. CloudWatch

Ce didacticiel utilise le runtime Node.js 24, mais nous avons également fourni des exemples de code dans d'autres langages d'exécution. Vous pouvez sélectionner l'onglet dans la zone suivante pour voir le code de l'exécution qui vous intéresse. Le JavaScript code que vous allez utiliser dans cette étape se trouve dans le premier exemple présenté dans l'JavaScriptonglet.

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Kinesis avec Lambda à l'aide de .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
```

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegrationSampleCode;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task FunctionHandler(KinesisEvent evnt, ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return;
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                throw;
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
```

```
        return data;
    }
}
```

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Kinesis avec Lambda à l'aide de Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent) error {
 if len(kinesisEvent.Records) == 0 {
  log.Printf("empty Kinesis event received")
  return nil
 }

 for _, record := range kinesisEvent.Records {
  log.Printf("processed Kinesis event with EventId: %v", record.EventID)
  recordDataBytes := record.Kinesis.Data
  recordDataText := string(recordDataBytes)
  log.Printf("record data: %v", recordDataText)
  // TODO: Do interesting work based on the new data
 }
 log.Printf("successfully processed %v records", len(kinesisEvent.Records))
```

```
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Kinesis avec Lambda à l'aide de Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;

public class Handler implements RequestHandler<KinesisEvent, Void> {
    @Override
    public Void handleRequest(final KinesisEvent event, final Context context) {
        LambdaLogger logger = context.getLogger();
        if (event.getRecords().isEmpty()) {
            logger.log("Empty Kinesis Event received");
            return null;
        }
        for (KinesisEvent.KinesisEventRecord record : event.getRecords()) {
            try {
                logger.log("Processed Event with EventId: "+record.getEventID());
                String data = new String(record.getKinesis().getData().array());
                logger.log("Data:"+ data);
```

```
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex) {
                logger.log("An error occurred:"+ex.getMessage());
                throw ex;
            }
        }
        logger.log("Successfully processed:"+event.getRecords().size()+"
 records");
        return null;
    }

}
```

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement Kinesis avec Lambda à l'aide de. JavaScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      throw err;
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
```

```
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

Utilisation d'un événement Kinesis avec Lambda à l'aide de. TypeScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      throw err;
    }
    logger.info(`Successfully processed ${event.Records.length} records.`);
  }
```

```
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Kinesis avec Lambda à l'aide de PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Kinesis\KinesisHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends KinesisHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
```

```
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleKinesis(KinesisEvent $event, Context $context): void
    {
        $this->logger->info("Processing records");
        $records = $event->getRecords();
        foreach ($records as $record) {
            $data = $record->getData();
            $this->logger->info(json_encode($data));
            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Kinesis avec Lambda à l'aide de Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import base64
```

```
def lambda_handler(event, context):

    for record in event['Records']:
        try:
            print(f"Processed Kinesis Event - EventID: {record['eventID']}")
            record_data = base64.b64decode(record['kinesis']
['data']).decode('utf-8')
            print(f"Record Data: {record_data}")
            # TODO: Do interesting work based on the new data
        except Exception as e:
            print(f"An error occurred {e}")
            raise e
    print(f"Successfully processed {len(event['Records'])} records.")
```

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Kinesis avec Lambda à l'aide de Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue => err
      $stderr.puts "An error occurred #{err}"
      raise err
    end
```

```
  end
  puts "Successfully processed #{event['Records'].length} records."
end

def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('UTF-8')
  # Placeholder for actual async work
  # You can use Ruby's asynchronous programming tools like async/await or fibers
 here.
  return data
end
```

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Kinesis avec Lambda à l'aide de Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::kinesis::KinesisEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) -> Result<(), Error>
 {
    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    event.payload.records.iter().for_each(|record| {
        tracing::info!("EventId:
 {}",record.event_id.as_deref().unwrap_or_default());

        let record_data = std::str::from_utf8(&record.kinesis.data);
```

```
        match record_data {
            Ok(data) => {
                // log the record data
                tracing::info!("Data: {}", data);
            }
            Err(e) => {
                tracing::error!("Error: {}", e);
            }
        }
    });

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

Pour créer la fonction

1. Créez un répertoire pour le projet, puis passez à ce répertoire.

   ```
   mkdir kinesis-tutorial
   cd kinesis-tutorial
   ```

2. Copiez l'exemple de JavaScript code dans un nouveau fichier nomméindex.js.

3. Créez un package de déploiement.

```
zip function.zip index.js
```

4. Créez une fonction Lambda à l'aide de la commande `create-function`.

```
aws lambda create-function --function-name ProcessKinesisRecords \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs24.x \
--role arn:aws:iam::111122223333:role/lambda-kinesis-role
```

Test de la fonction Lambda

Appelez votre fonction Lambda manuellement à l'aide de la commande `invoke` AWS Lambda CLI et d'un exemple d'événement Kinesis.

Pour tester la fonction Lambda

1. Copiez le code JSON suivant dans un fichier et enregistrez-le sous le nom `input.txt`.

```
{
    "Records": [
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692538361571095921575989136588898",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "approximateArrivalTimestamp": 1545084650.987
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::111122223333:role/lambda-kinesis-
role",
            "awsRegion": "us-east-2",
            "eventSourceARN": "arn:aws:kinesis:us-east-2:111122223333:stream/
lambda-stream"
        }
    ]
```

```
}
```

2. Utilisez la commande `invoke` pour envoyer l'événement à la fonction.

```
aws lambda invoke --function-name ProcessKinesisRecords \
--cli-binary-format raw-in-base64-out \
--payload file://input.txt outputfile.txt
```

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

La réponse est enregistrée dans `out.txt`.

Créez un flux Kinesis

Pour créer un flux, utilisez la commande `create-stream` .

```
aws kinesis create-stream --stream-name lambda-stream --shard-count 1
```

Exécutez la commande `describe-stream` suivante pour obtenir l'ARN du flux.

```
aws kinesis describe-stream --stream-name lambda-stream
```

Vous devriez voir la sortie suivante:

```
{
    "StreamDescription": {
        "Shards": [
            {
                "ShardId": "shardId-000000000000",
                "HashKeyRange": {
                    "StartingHashKey": "0",
                    "EndingHashKey": "340282366920746074317682119384634633455"
                },
                "SequenceNumberRange": {
                    "StartingSequenceNumber":
 "49591073947768692513481539594623130411957558361251844610"
```

```
                }
            }
        ],
        "StreamARN": "arn:aws:kinesis:us-east-1:111122223333:stream/lambda-stream",
        "StreamName": "lambda-stream",
        "StreamStatus": "ACTIVE",
        "RetentionPeriodHours": 24,
        "EnhancedMonitoring": [
            {
                "ShardLevelMetrics": []
            }
        ],
        "EncryptionType": "NONE",
        "KeyId": null,
        "StreamCreationTimestamp": 1544828156.0
    }
}
```

Vous utilisez l'ARN du flux à l'étape suivante pour associer le flux à la fonction Lambda.

Ajouter une source d'événement dans AWS Lambda

Exécutez la commande suivante AWS CLI `add-event-source`.

```
aws lambda create-event-source-mapping --function-name ProcessKinesisRecords \
--event-source  arn:aws:kinesis:us-east-1:111122223333:stream/lambda-stream \
--batch-size 100 --starting-position LATEST
```

Notez l'ID de mappage pour une utilisation ultérieure. Pour obtenir une liste des mappages de source d'événement, exécutez la commande suivante `list-event-source-mappings`.

```
aws lambda list-event-source-mappings --function-name ProcessKinesisRecords \
--event-source arn:aws:kinesis:us-east-1:111122223333:stream/lambda-stream
```

Dans la réponse, vous pouvez vérifier que la valeur d'état indique `enabled`. Les mappages de source d'événement peuvent être désactivés pour suspendre temporairement l'interrogation, ce qui entraîne la perte d'enregistrements.

Tester la configuration

Pour tester le mappage de source d'événement, ajoutez des enregistrements d'événements à votre flux Kinesis. La valeur `--data` est une chaîne que la commande CLI encode en base 64 avant de

l'envoyer à Kinesis. Vous pouvez exécuter la même commande plus d'une fois pour ajouter plusieurs enregistrements dans le flux.

```
aws kinesis put-record --stream-name lambda-stream --partition-key 1 \
--data "Hello, this is a test."
```

Lambda utilise le rôle d'exécution pour lire les enregistrements du flux. Ensuite, il invoque votre fonction Lambda en transmettant des lots d'enregistrements. La fonction décode les données de chaque enregistrement et les enregistre, en envoyant le résultat à CloudWatch Logs. Affichez les journaux dans la console CloudWatch .

Nettoyage de vos ressources

Vous pouvez maintenant supprimer les ressources que vous avez créées pour ce didacticiel, sauf si vous souhaitez les conserver. En supprimant AWS les ressources que vous n'utilisez plus, vous évitez des frais inutiles pour votre Compte AWS.

Pour supprimer le rôle d'exécution

1. Ouvrez la page Roles (Rôles) de la console IAM.
2. Sélectionnez le rôle d'exécution que vous avez créé.
3. Sélectionnez Delete (Supprimer).
4. Saisissez le nom du rôle dans le champ de saisie de texte et choisissez Delete (Supprimer).

Pour supprimer la fonction Lambda

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction que vous avez créée.
3. Sélectionnez Actions, Supprimer.
4. Saisissez **`confirm`** dans la zone de saisie de texte et choisissez Delete (Supprimer).

Pour supprimer le flux Kinesis

1. Connectez-vous à la console Kinesis AWS Management Console et ouvrez-la à https://console.aws.amazon.com l'adresse /kinesis.
2. Sélectionnez le flux que vous avez créé.
3. Sélectionnez Actions, Supprimer.

4. Saisissez **`delete`** dans le champ de saisie de texte.
5. Sélectionnez Supprimer.

Utilisation de Lambda avec Kubernetes

Vous pouvez déployer et gérer des fonctions Lambda avec l'API Kubernetes à l'aide d'AWS Controllers for Kubernetes (ACK) ou Crossplane.

AWS Controllers for Kubernetes (ACK)

Vous pouvez utiliser ACK pour déployer et gérer des ressources AWS de l'API Kubernetes. Par le biais d'ACK, AWS fournit des contrôleurs open source personnalisés pour des services AWS tels que Lambda, Amazon Elastic Container Registry (Amazon ECR), Amazon Simple Storage Service (Amazon S3) et Amazon SageMaker AI. Chaque service AWS pris en charge possède son propre contrôleur personnalisé. Dans votre cluster Kubernetes, installez un contrôleur pour chaque service AWS que vous souhaitez utiliser. Créez ensuite une définition de ressource personnalisée (Custom Resource Definition, CRD) pour définir les ressources AWS.

Nous vous recommandons d'utiliser Helm 3.8 ou version ultérieure pour installer les contrôleurs ACK. Chaque contrôleur ACK est doté de ses propres Charts de Helm, qui installent le contrôleur, les CRD et les règles RBAC de Kubernetes. Pour plus d'informations, consultez Installer un contrôleur ACK dans la documentation ACK.

Après avoir créé la ressource personnalisée ACK, vous pouvez l'utiliser comme n'importe quel autre objet Kubernetes intégré. Par exemple, vous pouvez déployer et gérer des fonctions Lambda avec vos chaînes d'outils Kubernetes préférées, notamment kubectl.

Voici quelques exemples de cas d'utilisation pour le provisionnement de fonctions Lambda via ACK :

- Votre organisation utilise le contrôle d'accès basé sur les rôles (RBAC) et des rôles IAM pour les comptes de service afin de créer les limites des autorisations. Avec ACK, vous pouvez réutiliser ce modèle de sécurité pour Lambda sans avoir à créer d'autres utilisateurs et stratégies.
- Votre organisation dispose d'un processus DevOps pour déployer des ressources dans un cluster Amazon Elastic Kubernetes Service (Amazon EKS) à l'aide de manifestes Kubernetes. Avec ACK, vous pouvez utiliser un manifeste pour provisionner des fonctions Lambda sans créer d'infrastructure distincte sous forme de modèles de code.

Pour plus d'informations sur l'utilisation d'ACK, consultez le didacticiel Lambda dans la documentation ACK.

Crossplane

Crossplane est un projet open source de la Cloud Native Computing Foundation (CNCF) qui utilise Kubernetes pour gérer les ressources de l'infrastructure cloud. Avec Crossplane, les développeurs peuvent demander une infrastructure sans avoir à en comprendre la complexité. Les équipes chargées de la plateforme gardent le contrôle de la manière dont l'infrastructure est provisionnée et gérée.

À l'aide de Crossplane, vous pouvez déployer et gérer des fonctions Lambda avec vos chaînes d'outils Kubernetes préférées, telles que kubectl, et tout pipeline CI/CD capable de déployer des manifestes sur Kubernetes. Voici quelques exemples de cas d'utilisation pour le provisionnement de fonctions Lambda via Crossplane :

- Votre entreprise souhaite renforcer la conformité en s'assurant que les fonctions Lambda disposent des balises appropriées. Les équipes chargées de la plateforme peuvent utiliser Compositions Crossplane pour définir cette stratégie par le biais d'abstractions d'API. Les développeurs peuvent ensuite utiliser ces abstractions pour déployer des fonctions Lambda avec des balises.
- Votre projet utilise GitOps avec Kubernetes. Dans ce modèle, Kubernetes réconcilie en permanence le référentiel Git (état souhaité) avec les ressources qui s'exécutent à l'intérieur du cluster (état actuel). S'il existe des différences, le processus GitOps apporte automatiquement des modifications au cluster. Vous pouvez utiliser GitOps avec Kubernetes pour déployer et gérer des fonctions Lambda via Crossplane, à l'aide d'outils et de concepts Kubernetes bien connus tels que des CRD et des contrôleurs.

Pour en savoir plus sur l'utilisation de Crossplane avec Lambda, consultez les rubriques suivantes :

- Plans AWS pour Crossplane : ce référentiel inclut des exemples d'utilisation de Crossplane pour déployer des ressources AWS, y compris les fonctions Lambda.

  > **Note**
  >
  > Les plans AWS pour Crossplane sont en cours de développement et ne devraient pas être utilisés en production.

- Déploiement de Lambda avec Amazon EKS et Crossplane : cette vidéo montre un exemple avancé de déploiement d'une architecture sans serveur AWS avec Crossplane, explorant la conception du point de vue du développeur et de la plateforme.

Utilisation de Lambda avec Amazon MQ

> **Note**
>
> Si vous souhaitez envoyer des données à une cible autre qu'une fonction Lambda ou enrichir les données avant de les envoyer, consultez Amazon EventBridge Pipes.

Amazon MQ est un service d'agent de messages géré pour Apache ActiveMQ et RabbitMQ. Un agent de messages permet à des applications et composants logiciels de communiquer à l'aide de divers langages de programmation, systèmes d'exploitation et autres protocoles de messagerie formels par le biais de rubriques ou d'événements en file d'attente.

Amazon MQ peut également gérer des instances Amazon Elastic Compute Cloud EC2 (Amazon) en votre nom en installant des courtiers ActiveMQ ou RabbitMQ et en fournissant différentes topologies de réseau et d'autres besoins en infrastructure.

Vous pouvez utiliser une fonction Lambda pour traiter des enregistrements de votre agent de messages Amazon MQ. Lambda invoque votre fonction via un mappage de source d'événement, une ressource Lambda qui lit les messages de votre agent et invoque la fonction de manière synchrone.

> **Warning**
>
> Les mappages des sources d'événements Lambda traitent chaque événement au moins une fois, et le traitement des enregistrements peut être dupliqué. Pour éviter les problèmes potentiels liés à des événements dupliqués, nous vous recommandons vivement de rendre votre code de fonction idempotent. Pour en savoir plus, consultez Comment rendre ma fonction Lambda idempotente dans le Knowledge Center. AWS

Le mappage de source d'événement Amazon MQ est sujet aux restrictions de configuration suivantes :

- Simultanéité : les fonctions Lambda qui utilisent un mappage des sources d'événements Amazon MQ disposent d'un paramètre de simultanéité maximale par défaut. Pour ActiveMQ, le service Lambda limite le nombre d'environnements d'exécution simultanés à cinq par mappage des sources d'événements Amazon MQ. Pour RabbitMQ, le nombre d'environnements d'exécution simultanés est limité à 1 par mappage des sources d'événements Amazon MQ. Même si vous

modifiez les paramètres de simultanéité réservés ou fournis de votre fonction, le service Lambda ne mettra pas plus d'environnements d'exécution à disposition. Pour demander une augmentation de la simultanéité maximale par défaut pour un seul mappage de source d'événement Amazon MQ, Support contactez l'UUID du mappage de source d'événement, ainsi que la région. Étant donné que les augmentations sont appliquées au niveau de chaque mappage des sources d'événements, et non au niveau du compte ou de la région, vous devez demander manuellement une augmentation de mise à l'échelle pour chaque mappage des sources d'événements.

- Traitement entre comptes – Lambda ne prend pas en charge le traitement entre comptes. Vous ne pouvez pas utiliser Lambda pour traiter des enregistrements d'un agent de messages Amazon MQ se trouvant dans un autre Compte AWS.
- Authentification — Pour ActiveMQ, seul ActiveMQ est pris en charge. SimpleAuthenticationPlugin Pour RabbitMQ, seule l'authentification PLAIN est prise en charge. Les utilisateurs doivent utiliser AWS Secrets Manager pour gérer leurs informations d'identification. Pour plus d'informations sur l'authentification ActiveMQ, consultez Intégration d'agents ActiveMQ avec LDAP dans le Guide du développeur Amazon MQ.
- Quota de connexion – Les agents ont un nombre maximum de connexions autorisées par protocole de niveau filaire. Ce quota est basé sur le type d'instance de l'agent. Pour plus d'informations, consultez la section Agents de Quotas dans Amazon dans le Manuel du développeur Amazon MQ.
- Connectivité – Vous pouvez créer des agents dans un cloud privé virtuel (VPC) public ou privé. Pour le mode privé VPCs, votre fonction Lambda doit accéder au VPC pour recevoir des messages. Pour plus d'informations, consultez the section called "Configurer la sécurité réseau", plus loin dans cette section.
- Destinations d'événements – Seules les destinations en file d'attente sont prises en charge. Toutefois, vous pouvez utiliser une rubrique virtuelle qui se comporte comme une rubrique en interne tout en interagissant avec Lambda en tant que file d'attente. Pour plus d'informations, consultez Virtual Destinations (Destinations virtuelles) sur le site web Apache ActiveMQ, et Virtual Hosts (Hôtes virtuels) sur le site web RabbitMQ.
- Topologie réseau – Pour ActiveMQ, un mappage de source d'événement prend en charge une seule instance ou un seul agent en veille. Pour RabbitMQ, un mappage de source d'événement prend en charge un seul agent d'instance ou un seul déploiement de cluster. Les agents à instance unique nécessitent un point de terminaison de basculement. Pour plus d'informations sur ces modes de déploiement d'agent, consultez Architecture d'agent ActiveMQ et Architecture d'agent RabbitMQ dans le Guide du développeur Amazon MQ.
- Protocoles – Les protocoles pris en charge dépendent du type d'intégration d'Amazon MQ.

- Pour les intégrations ActiveMQ, Lambda consomme les messages à l'aide OpenWire/Java du protocole Message Service (JMS). Aucun autre protocole n'est pris en charge pour la consommation de messages. Dans le protocole JMS, seuls `TextMessage` et `BytesMessage` sont pris en charge. Lambda prend également en charge les propriétés personnalisées JMS. Pour plus d'informations sur le OpenWire protocole, consultez OpenWirele site Web d'Apache ActiveMQ.
- Pour les intégrations RabbitMQ, Lambda consomme des messages à l'aide du protocole AMQP 0-9-1. Aucun autre protocole n'est pris en charge pour la consommation de messages. Pour plus d'informations sur l'implémentation par RabbitMQ du protocole AMQP 0-9-1, consultez Guide de référence complet AMQP 0-9-1 sur le site web de RabbitMQ.

Lambda prend automatiquement en charge les dernières versions d'ActiveMQ et de RabbitMQ qu'Amazon MQ prend en charge. Pour connaître les dernières versions prises en charge, consultez Notes de mise à jour Amazon MQ dans le Guide du développeur Amazon MQ.

> **Note**
>
> Par défaut, Amazon MQ comporte une fenêtre de maintenance hebdomadaire pour les agents. Pendant cette période, les agents ne sont pas disponibles. Pendant ce temps, Lambda ne peut pas traiter les messages des agents sans veille.

Rubriques

- Comprendre le groupe de consommateurs Lambda pour Amazon MQ
- Configuration de source d'événements Amazon MQ pour Lambda
- Paramètres de mappage des sources d'événements
- Filtrer les événement à partir d'une source d'événements Amazon MQ
- Résoudre les erreurs de mappage des sources d'événement Amazon MQ

Comprendre le groupe de consommateurs Lambda pour Amazon MQ

Pour interagir avec Amazon MQ, Lambda crée un groupe de consommateurs qui peut être lu à partir de vos agents Amazon MQ. Le groupe de consommateurs est créé avec le même ID qu'un mappage de source d'événements UUID.

Pour les sources d'événements Amazon MQ, Lambda réunit les enregistrements par lots et les envoie à votre fonction dans une seule charge utile. Pour contrôler le comportement, vous pouvez configurer la fenêtre de traitement par lots et la taille du lot. Lambda extrait les messages jusqu'à ce qu'il traite la taille maximale de la charge utile de 6 Mo, que la fenêtre de traitement par lots expire ou que le nombre d'enregistrements atteigne la taille totale du lot. Pour de plus amples informations, veuillez consulter Comportement de traitement par lots.

Le groupe de consommateurs récupère les messages sous forme de BLOB d'octets, puis les code en base64 dans une charge unique JSON puis invoque votre fonction. Si votre fonction renvoie une erreur pour l'un des messages d'un lot, Lambda réessaie le lot de messages complet jusqu'à ce que le traitement réussisse ou que les messages expirent.

> **Note**
>
> Alors que les fonctions Lambda ont généralement un délai d'expiration maximal de 15 minutes, les mappages des sources d'événement pour Amazon MSK, Apache Kafka autogéré, Amazon DocumentDB et Amazon MQ pour ActiveMQ et RabbitMQ ne prennent en charge que les fonctions dont le délai d'expiration maximal est de 14 minutes. Cette contrainte garantit que le mappage des sources d'événements peut gérer correctement les erreurs de fonction et effectuer de nouvelles tentatives.

Vous pouvez surveiller l'utilisation simultanée d'une fonction donnée à l'aide de la `ConcurrentExecutions` métrique d'Amazon CloudWatch. Pour de plus amples informations sur la simultanéité, consultez the section called "Configuration de la simultanéité réservée".

Example Événements d'enregistrement Amazon MQ

ActiveMQ

```
{
   "eventSource": "aws:mq",
   "eventSourceArn": "arn:aws:mq:us-
east-2:111122223333:broker:test:b-9bcfa592-423a-4942-879d-eb284b418fc8",
   "messages": [
      {
        "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-
east-2.amazonaws.com-37557-1234520418293-4:1:1:1:1",
        "messageType": "jms/text-message",
        "deliveryMode": 1,
```

```
        "replyTo": null,
        "type": null,
        "expiration": "60000",
        "priority": 1,
        "correlationId": "myJMSCoID",
        "redelivered": false,
        "destination": {
          "physicalName": "testQueue"
        },
        "data":"QUJDOkFBQUE=",
        "timestamp": 1598827811958,
        "brokerInTime": 1598827811958,
        "brokerOutTime": 1598827811959,
        "properties": {
          "index": "1",
          "doAlarm": "false",
          "myCustomProperty": "value"
        }
      },
      {
        "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-
east-2.amazonaws.com-37557-1234520418293-4:1:1:1:1",
        "messageType": "jms/bytes-message",
        "deliveryMode": 1,
        "replyTo": null,
        "type": null,
        "expiration": "60000",
        "priority": 2,
        "correlationId": "myJMSCoID1",
        "redelivered": false,
        "destination": {
          "physicalName": "testQueue"
        },
        "data":"LQaGQ82S48k=",
        "timestamp": 1598827811958,
        "brokerInTime": 1598827811958,
        "brokerOutTime": 1598827811959,
        "properties": {
          "index": "1",
          "doAlarm": "false",
          "myCustomProperty": "value"
        }
      }
   ]
```

```
}
```

RabbitMQ

```
{
  "eventSource": "aws:rmq",
  "eventSourceArn": "arn:aws:mq:us-
east-2:111122223333:broker:pizzaBroker:b-9bcfa592-423a-4942-879d-eb284b418fc8",
  "rmqMessagesByQueue": {
    "pizzaQueue::/": [
      {
        "basicProperties": {
          "contentType": "text/plain",
          "contentEncoding": null,
          "headers": {
            "header1": {
              "bytes": [
                118,
                97,
                108,
                117,
                101,
                49
              ]
            },
            "header2": {
              "bytes": [
                118,
                97,
                108,
                117,
                101,
                50
              ]
            },
            "numberInHeader": 10
          },
          "deliveryMode": 1,
          "priority": 34,
          "correlationId": null,
          "replyTo": null,
          "expiration": "60000",
```

```
          "messageId": null,
          "timestamp": "Jan 1, 1970, 12:33:41 AM",
          "type": null,
          "userId": "AIDACKCEVSQ6C2EXAMPLE",
          "appId": null,
          "clusterId": null,
          "bodySize": 80
        },
        "redelivered": false,
        "data": "eyJ0aW1lb3V0IjowLCJkYXRhIjoiQ1pybWYwR3c4T3Y0YnFMUXhENEUifQ=="
      }
    ]
  }
}
```

> **Note**
>
> Dans l'exemple RabbitMQ, `pizzaQueue` est le nom de la file d'attente RabbitMQ, et `/` est le nom de l'hôte virtuel. Lors de la réception de messages, la source d'événement répertorie les messages sous `pizzaQueue::/`.

Configuration de source d'événements Amazon MQ pour Lambda

Rubriques

- Configurer la sécurité réseau
- Créer le mappage des sources d'événements

Configurer la sécurité réseau

Pour donner à Lambda un accès complet à Amazon MQ via votre mappage des sources d'événements, soit votre agent doit utiliser un point de terminaison public (adresse IP publique), soit vous devez fournir un accès au VPC Amazon dans lequel vous avez créé l'agent.

Lorsque vous utilisez Amazon MQ avec Lambda, créez des points de terminaison de VPC AWS PrivateLink qui permettent à votre fonction d'accéder aux ressources de votre Amazon VPC.

> **Note**
>
> AWS PrivateLink Les points de terminaison VPC sont requis pour les fonctions avec des mappages de sources d'événements qui utilisent le mode par défaut (à la demande) pour les sondeurs d'événements. Si le mappage de votre source d'événements utilise le mode provisionné, vous n'avez pas besoin de configurer les points de terminaison AWS PrivateLink VPC.

Créez un point de terminaison pour accéder aux ressources suivantes :

- Lambda — Créez un point de terminaison pour le principal de service Lambda.
- AWS STS — Créez un point de terminaison pour le AWS STS afin qu'un directeur de service assume un rôle en votre nom.
- Secrets Manager : si votre agent utilise Secrets Manager pour stocker les informations d'identification, créez un point de terminaison pour Secrets Manager.

Vous pouvez également configurer une passerelle NAT sur chaque sous-réseau public d'Amazon VPC. Pour de plus amples informations, veuillez consulter the section called "Accès Internet pour les fonctions VPC".

Lorsque vous créez un mappage de sources d'événements pour Amazon MQ, Lambda vérifie si des interfaces réseau élastiques (ENIs) sont déjà présentes pour les sous-réseaux et les groupes de sécurité configurés pour votre Amazon VPC. Si Lambda trouve des objets existants ENIs, il essaie de les réutiliser. Sinon, Lambda en crée un nouveau ENIs pour se connecter à la source de l'événement et appeler votre fonction.

> **Note**
>
> Les fonctions Lambda s'exécutent toujours au sein VPCs du service Lambda. La configuration VPC de votre fonction n'affecte pas le mappage des sources d'événements. Seule la configuration réseau des sources d'événements détermine la manière dont Lambda se connecte à votre source d'événements.

Configurez les groupes de sécurité pour le VPC Amazon contenant votre agent. Par défaut, Amazon MQ utilise les ports suivants : 61617 (Amazon MQ pour ActiveMQ) 5671 et (Amazon MQ pour RabbitMQ).

- Règles entrantes – Autorisent tout le trafic sur le port de l'agent par défaut pour le groupe de sécurité associé à votre source d'événement. Vous pouvez également utiliser une règle de groupe de sécurité avec référence circulaire pour autoriser l'accès à partir d'instances appartenant au même groupe de sécurité.
- Règles de sortie : autorisez tout le trafic sur le port 443 pour les destinations externes si votre fonction doit communiquer avec les AWS services. Vous pouvez également utiliser une règle de groupe de sécurité à référencement automatique pour limiter l'accès au courtier si vous n'avez pas besoin de communiquer avec d'autres AWS services.
- Règles entrantes relatives au point de terminaison Amazon VPC – Si vous utilisez un point de terminaison Amazon VPC, le groupe de sécurité associé à votre point de terminaison Amazon VPC doit autoriser le trafic entrant sur le port 443 en provenance du groupe de sécurité de l'agent.

Si votre agent utilise l'authentification, vous pouvez également restreindre la politique de point de terminaison pour le point de terminaison Secrets Manager. Pour appeler l'API Secrets Manager, Lambda utilise votre rôle de fonction, et non le principal de service Lambda.

Example Politique de point de terminaison de VPC — Point de terminaison Secrets Manager

```
{
      "Statement": [
          {
              "Action": "secretsmanager:GetSecretValue",
              "Effect": "Allow",
              "Principal": {
                  "AWS": [
                      "arn:aws::iam::123456789012:role/my-role"
                  ]
              },
              "Resource": "arn:aws::secretsmanager:us-west-2:123456789012:secret:my-
secret"
          }
      ]
  }
```

Lorsque vous utilisez des points de terminaison Amazon VPC, AWS achemine vos appels d'API pour appeler votre fonction à l'aide de l'Elastic Network Interface (ENI) du point de terminaison. Le directeur du service Lambda doit faire appel à tous `lambda:InvokeFunction` les rôles et fonctions qui les utilisent. ENIs

Par défaut, les points de terminaison Amazon VPC disposent de politiques IAM ouvertes qui permettent un accès étendu aux ressources. La meilleure pratique consiste à restreindre ces politiques pour effectuer les actions nécessaires à l'aide de ce point de terminaison. Pour garantir que votre mappage des sources d'événements est en mesure d'invoquer votre fonction Lambda, la politique de point de terminaison de VPC doit autoriser le principal de service Lambda à appeler `sts:AssumeRole` et `lambda:InvokeFunction`. Le fait de restreindre vos politiques de point de terminaison de VPC pour autoriser uniquement les appels d'API provenant de votre organisation empêche le mappage des sources d'événements de fonctionner correctement. C'est pourquoi `"Resource": "*"` est requis dans ces politiques.

Les exemples de politiques de point de terminaison de VPC suivants montrent comment accorder l'accès requis au principal de service Lambda pour les points de terminaison AWS STS et Lambda.

Example Politique de point de terminaison VPC — point de terminaison AWS STS

```
{
      "Statement": [
          {
              "Action": "sts:AssumeRole",
              "Effect": "Allow",
              "Principal": {
                  "Service": [
                      "lambda.amazonaws.com"
                  ]
              },
              "Resource": "*"
          }
      ]
    }
```

Example Politique de point de terminaison de VPC — Point de terminaison Lambda

```
{
      "Statement": [
          {
              "Action": "lambda:InvokeFunction",
```

```
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Créer le mappage des sources d'événements

Créez un mappage de source d'événement pour indiquer à Lambda d'envoyer des enregistrements d'un agent Amazon MQ à une fonction Lambda. Vous pouvez créer plusieurs mappages de source d'événement pour traiter les mêmes données avec plusieurs fonctions, ou pour traiter des éléments à partir de plusieurs sources avec une seule fonction.

Pour configurer votre fonction afin qu'elle lise à partir d'Amazon MQ, ajoutez les autorisations requises et créez un déclencheur MQ dans la console Lambda.

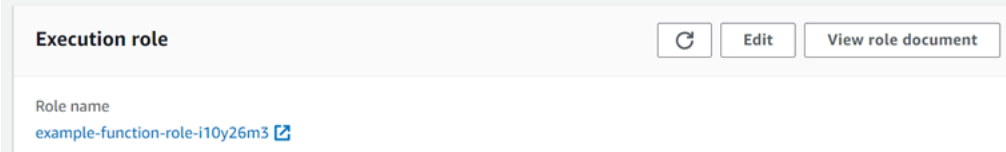
Pour lire les enregistrements d'un agent Amazon MQ, votre fonction Lambda a besoin des autorisations suivantes. Vous autorisez Lambda à interagir avec votre agent Amazon MQ et ses ressources sous-jacentes en ajoutant des déclarations d'autorisation à votre rôle d'exécution de fonction :

- mq : DescribeBroker
- responsable des secrets : GetSecretValue
- EC2 : CreateNetworkInterface
- EC2 : DeleteNetworkInterface
- EC2 : DescribeNetworkInterfaces
- EC2 : DescribeSecurityGroups
- EC2 : DescribeSubnets
- EC2 : DescribeVpcs
- journaux : CreateLogGroup
- journaux : CreateLogStream
- journaux : PutLogEvents

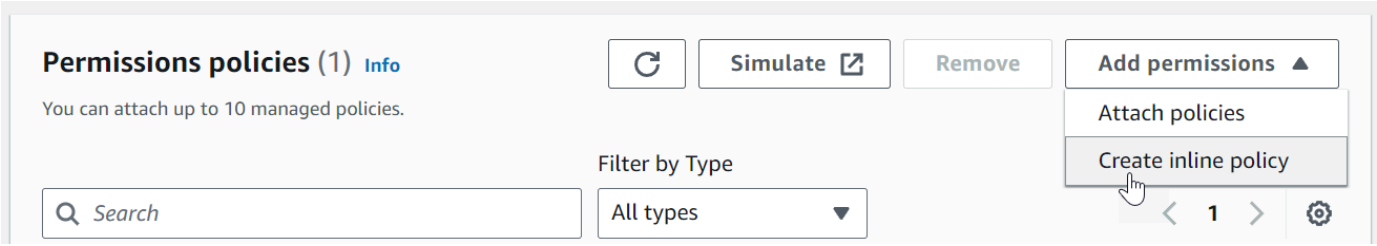
> **Note**
>
> Lorsque vous utilisez une clé gérée par le client chiffrée, ajoutez également l'autorisation `kms:Decrypt`.

Pour ajouter des autorisations et créer un déclencheur

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom d'une fonction.
3. Choisissez l'onglet Configuration, puis Permissions (Autorisations).
4. Sous Nom du rôle, cliquez sur le lien vers votre rôle d'exécution. Ce lien ouvre le rôle dans la console IAM.



5. Sélectionnez Ajouter des autorisations, puis Ajouter la politique.



6. Dans l'Éditeur de politique, sélectionnez JSON. Saisissez la politique suivante. Votre fonction a besoin de ces autorisations pour lire à partir d'un agent Amazon MQ.

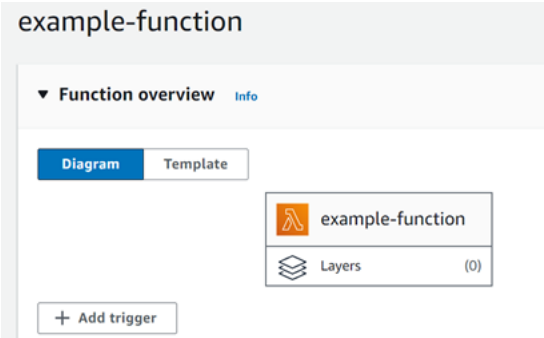
   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
          "mq:DescribeBroker",
          "secretsmanager:GetSecretValue",
          "ec2:CreateNetworkInterface",
```

```
            "ec2:DeleteNetworkInterface",
            "ec2:DescribeNetworkInterfaces",
            "ec2:DescribeSecurityGroups",
            "ec2:DescribeSubnets",
            "ec2:DescribeVpcs",
            "logs:CreateLogGroup",
            "logs:CreateLogStream",
            "logs:PutLogEvents"
        ],
        "Resource": "*"
    }
  ]
}
```

> **Note**
>
> Lorsque vous utilisez une clé gérée par le client chiffrée, vous devez également ajouter l'autorisation `kms:Decrypt`.

7. Choisissez Suivant. Saisissez un nom de politique, puis choisissez Créer une politique.
8. Revenez à votre fonction dans la console Lambda. Sous Function overview (Vue d'ensemble de la fonction), choisissez Add trigger (Ajouter un déclencheur).



9. Choisissez le type de déclencheur MQ.
10. Configurez les options requises, puis choisissez Add (Ajouter).

Lambda prend en charge les options suivantes pour les sources d'événement Amazon MQ.

- MQ broker (Agent MQ) – Sélectionnez un agent Amazon MQ.
- Batch size (Taille de lot) – Définissez le nombre maximum de messages à extraire dans un lot.

- Queue name (Nom de file d'attente) – Entrez la file d'attente Amazon MQ à consommer.
- Source access configuration (Configuration de l'accès source) – Entrez les informations d'hôte virtuel et le secret Secrets Manager qui stocke vos informations d'identification d'agent.
- Enable trigger (Activer le déclencheur) – Désactivez le déclencheur pour arrêter le traitement des enregistrements.

Pour activer ou désactiver le déclencheur (ou le supprimer), choisissez le déclencheur MQ dans le concepteur. Pour reconfigurer le déclencheur, utilisez les opérations d'API de mappage de source d'événement.

Paramètres de mappage des sources d'événements

Tous les types de sources d'événement Lambda partagent les mêmes opérations d'API CreateEventSourceMapping et UpdateEventSourceMapping. Cependant, seuls certains paramètres s'appliquent à Amazon MQ et à RabbitMQ.

Paramètre	Obligatoire	Par défaut	Remarques
BatchSize	N	100	Maximum : 10 000.
Activé	N	VRAI	none
FunctionName	Y	N/A	none
FilterCriteria	N	N/A	Contrôle des événements envoyés par Lambda à votre fonction
MaximumBatchingWindowInSeconds	N	500 ms	Comportement de traitement par lots
Files d'attente	N	N/A	Le nom de la file d'attente de destination de l'agent Amazon MQ à consommer.

Paramètre	Obligatoire	Par défaut	Remarques
SourceAccessConfigurations	N	N/A	Pour ActiveMQ, informations d'identification BASIC_AUTH. Pour RabbitMQ, il peut contenir à la fois des informations d'identification BASIC_AUTH et des informations VIRTUAL_HOST.

Filtrer les événement à partir d'une source d'événements Amazon MQ

Vous pouvez utiliser le filtrage d'événements pour contrôler les enregistrements d'un flux ou d'une file d'attente que Lambda envoie à votre fonction. Pour obtenir des informations générales sur le fonctionnement du filtrage des événements, consultez the section called "Filtrage des événements".

Cette section porte sur le filtrage des événements pour les sources Amazon MQ.

> **Note**
>
> Les mappages des sources d'événements Amazon MQ prennent uniquement en charge le filtrage sur la clé `data`.

Rubriques

- Principes de base du filtrage des événements Amazon MQ

Principes de base du filtrage des événements Amazon MQ

Supposons que votre file d'attente de messages Amazon MQ contienne des messages au format JSON valide ou sous forme de chaînes simples. Un exemple d'enregistrement ressemblerait à ce qui suit, avec les données converties en chaîne encodée en Base64 dans le champ `data`.

ActiveMQ

```
{
    "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-
east-2.amazonaws.com-37557-1234520418293-4:1:1:1:1",
    "messageType": "jms/text-message",
    "deliveryMode": 1,
    "replyTo": null,
    "type": null,
    "expiration": "60000",
    "priority": 1,
    "correlationId": "myJMSCoID",
    "redelivered": false,
    "destination": {
      "physicalName": "testQueue"
    },
    "data":"QUJDOkFBQUE=",
    "timestamp": 1598827811958,
    "brokerInTime": 1598827811958,
    "brokerOutTime": 1598827811959,
    "properties": {
      "index": "1",
      "doAlarm": "false",
      "myCustomProperty": "value"
    }
}
```

RabbitMQ

```
{
    "basicProperties": {
        "contentType": "text/plain",
        "contentEncoding": null,
        "headers": {
            "header1": {
                "bytes": [
                  118,
                  97,
                  108,
                  117,
                  101,
                  49
                ]
```

```
            },
            "header2": {
                "bytes": [
                  118,
                  97,
                  108,
                  117,
                  101,
                  50
                ]
            },
            "numberInHeader": 10
        },
        "deliveryMode": 1,
        "priority": 34,
        "correlationId": null,
        "replyTo": null,
        "expiration": "60000",
        "messageId": null,
        "timestamp": "Jan 1, 1970, 12:33:41 AM",
        "type": null,
        "userId": "AIDACKCEVSQ6C2EXAMPLE",
        "appId": null,
        "clusterId": null,
        "bodySize": 80
        },
    "redelivered": false,
    "data": "eyJ0aW1lb3V0IjowLCJkYXRhIjoiQ1pybWYwR3c4T3Y0YnFMUXhENEUifQ=="
}
```

Pour les agents Active MQ et Rabbit MQ, vous pouvez utiliser le filtrage d'événements pour filtrer les enregistrements à l'aide de la clé `data`. Supposons que votre file d'attente Amazon MQ contienne des messages au format JSON suivant.

```
{
    "timeout": 0,
    "IPAddress": "203.0.113.254"
}
```

Pour filtrer uniquement les enregistrements dont le champ `timeout` est supérieur à 0, l'objet `FilterCriteria` serait le suivant.

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : { \"timeout\" : [ { \"numeric\": [ \">\",
 0] } } ] } }"
        }
    ]
}
```

Pour plus de clarté, voici la valeur du `Pattern` de filtre étendu en JSON simple :

```
{
    "data": {
        "timeout": [ { "numeric": [ ">", 0 ] } ]
        }
}
```

Vous pouvez ajouter votre filtre à l'aide de la console, d'AWS CLI ou d'un modèle AWS SAM.

Console

pour ajouter ce filtre à l'aide de la console, suivez les instructions de Attacher des critères de filtre à un mappage de sources d'événements (console) et saisissez la chaîne suivante comme critère de filtre.

```
{ "data" : { "timeout" : [ { "numeric": [ ">", 0 ] } ] } }
```

AWS CLI

Pour créer un nouveau mappage des sources d'événements avec ces critères de filtrage à l'aide de l'outil AWS Command Line Interface (AWS CLI), exécutez la commande suivante.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:mq:us-east-2:123456789012:broker:my-
broker:b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"timeout\" :
 [ { \"numeric\": [ \">\", 0 ] } ] } }"}]}'
```

Pour ajouter ces critères de filtre à un mappage des sources d'événements existant, exécutez la commande suivante.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"timeout\" :
 [ { \"numeric\": [ \">\", 0 ] } ] } }"}]}'
```

Pour ajouter ces critères de filtre à un mappage des sources d'événements existant, exécutez la commande suivante.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"timeout\" :
 [ { \"numeric\": [ \">\", 0 ] } ] } }"}]}'
```

AWS SAM

Pour ajouter ce filtre à l'aide d'AWS SAM, ajoutez l'extrait suivant au modèle YAML de votre source d'événement.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "data" : { "timeout" : [ { "numeric": [ ">", 0 ] } ] } }'
```

Avec Amazon MQ, vous pouvez également filtrer les enregistrements dont le message est une chaîne simple. Supposons que vous vouliez traiter uniquement les enregistrements dont le message commence par « Result: ». L'objet `FilterCriteria` se présente comme suit.

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : [ { \"prefix\": \"Result: \" } ] }"
        }
    ]
}
```

Pour plus de clarté, voici la valeur du `Pattern` de filtre étendu en JSON simple :

```
{
    "data": [
        {
```

```
        "prefix": "Result: "
        }
    ]
}
```

Vous pouvez ajouter votre filtre à l'aide de la console, d'AWS CLI ou d'un modèle AWS SAM.

Console

Pour ajouter ce filtre à l'aide de la console, suivez les instructions de Attacher des critères de filtre à un mappage de sources d'événements (console) et saisissez la chaîne suivante comme critère de filtre.

```
{ "data" : [ { "prefix": "Result: " } ] }
```

AWS CLI

Pour créer un nouveau mappage des sources d'événements avec ces critères de filtrage à l'aide de l'outil AWS Command Line Interface (AWS CLI), exécutez la commande suivante.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:mq:us-east-2:123456789012:broker:my-
broker:b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : [ { \"prefix\":
 \"Result: \" } ] }"}]}'
```

Pour ajouter ces critères de filtre à un mappage des sources d'événements existant, exécutez la commande suivante.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : [ { \"prefix\":
 \"Result: \" } ] }"}]}'
```

AWS SAM

Pour ajouter ce filtre à l'aide d'AWS SAM, ajoutez l'extrait suivant au modèle YAML de votre source d'événement.

```
FilterCriteria:
```

```
  Filters:
    - Pattern: '{ "data" : [ { "prefix": "Result " } ] }'
```

Les messages Amazon MQ doivent être des chaînes codées en UTF-8, soit des chaînes en texte brut, soit au format JSON. En effet, Lambda décode les tableaux d'octets Amazon MQ en UTF-8 avant d'appliquer des critères de filtre. Si vos messages utilisent un autre encodage, tel que UTF-16 ou ASCII, ou si le format du message ne correspond pas au format `FilterCriteria`, Lambda traite uniquement les filtres de métadonnées. Le tableau suivant résume le comportement spécifique :

Format du du message entrant	Modèle de filtre de format pour les propriétés des messages	Action obtenue.
Chaîne de texte brut	Chaîne de texte brut	Lambda filtre en fonction de vos critères de filtre.
Chaîne de texte brut	Pas de modèle de filtre pour les propriétés des données	Lambda filtre (uniquement selon les autres propriétés de métadonnées) en fonction de vos critères de filtre.
Chaîne de texte brut	JSON valide	Lambda filtre (uniquement selon les autres propriétés de métadonnées) en fonction de vos critères de filtre.
JSON valide	Chaîne de texte brut	Lambda filtre (uniquement selon les autres propriétés de métadonnées) en fonction de vos critères de filtre.
JSON valide	Pas de modèle de filtre pour les propriétés des données	Lambda filtre (uniquement selon les autres propriétés de métadonnées) en fonction de vos critères de filtre.
JSON valide	JSON valide	Lambda filtre en fonction de vos critères de filtre.

Format du du message entrant	Modèle de filtre de format pour les propriétés des messages	Action obtenue.
Chaîne non codée UTF-8	JSON, chaîne de texte brut ou aucun modèle	Lambda filtre (uniquement selon les autres propriétés de métadonnées) en fonction de vos critères de filtre.

Résoudre les erreurs de mappage des sources d'événement Amazon MQ

Quand une fonction Lambda rencontre une erreur irrécupérable, votre consommateur Amazon MQ arrête le traitement des enregistrements. D'autres consommateurs peuvent continuer le traitement, à condition qu'ils ne rencontrent pas la même erreur. Pour déterminer la cause potentielle d'un consommateur arrêté, vérifiez le champ `StateTransitionReason` dans les détails de retour de votre `EventSourceMapping` pour l'un des codes suivants :

`ESM_CONFIG_NOT_VALID`

La configuration de mappage de source d'événement n'est pas valide.

`EVENT_SOURCE_AUTHN_ERROR`

Lambda n'a pas réussi à authentifier la source de l'événement.

`EVENT_SOURCE_AUTHZ_ERROR`

Lambda ne dispose pas des autorisations requises pour accéder à la source d'événement.

`FUNCTION_CONFIG_NOT_VALID`

La configuration de la fonction n'est pas valide.

Les enregistrements ne sont pas non plus traités si Lambda les abandonne en raison de leur taille. La taille limite pour les enregistrements Lambda est de 6 Mo. Pour relivrer des messages en cas d'erreur de fonction, vous pouvez utiliser une file d'attente de lettres mortes (DLQ). Pour plus d'informations, consultez Message Redelivery and DLQ Handling (Relivraison des messages et gestion des DLQ), et le Guide de fiabilité sur le site web RabbitMQ.

> **Note**
>
> Lambda ne prend pas en charge les stratégies de relivraison personnalisées. Au lieu de cela, Lambda utilise une stratégie avec les valeurs par défaut de la page Stratégie de relivraison sur le site web d'Apache ActiveMQ, avec `maximumRedeliveries` défini sur 6.

Utilisation AWS Lambda avec Amazon RDS

Vous pouvez connecter une fonction Lambda à une base de données Amazon Relational Database Service (Amazon RDS) directement via un proxy Amazon RDS. Les connexions directes sont utiles dans les scénarios simples, et les proxys sont recommandés pour la production. Un proxy de base de données gère un groupe de connexions partagées à la base de données qui permet à votre fonction d'atteindre des niveaux de simultanéité élevés sans épuiser les connexions de base de données.

Nous recommandons d'utiliser Proxy Amazon RDS pour les fonctions Lambda qui effectuent fréquemment de brèves connexions à la base de données, ou qui ouvrent et ferment un grand nombre de connexions à la base de données. Pour de plus amples informations, veuillez consulter la rubrique Automatically connecting a Lambda function and a DB instance du Guide du développeur Amazon Relational Database Service.

> **Tip**
>
> Pour connecter rapidement une fonction Lambda à une base de données Amazon RDS, vous pouvez utiliser l'assistant guidé intégré à la console. Procédez comme suit pour ouvrir l'assistant :
>
> 1. Ouvrez la page Functions (Fonctions) de la console Lambda.
> 2. Sélectionnez la fonction à laquelle vous souhaitez connecter une base de données.
> 3. Dans l'onglet Configuration, sélectionnez Bases de données RDS.
> 4. Choisissez Se connecter à la base de données RDS.
>
> Après avoir connecté votre fonction à une base de données, vous pouvez créer un proxy en choisissant Ajouter un proxy.

Configuration de votre fonction pour qu'elle fonctionne avec les ressources RDS

Dans la console Lambda, vous pouvez allouer et configurer des instances de base de données et des ressources de proxy Amazon RDS. Vous pouvez le faire en accédant aux Bases de données RDS sous l'onglet Configuration. Vous pouvez également créer et configurer des connexions aux fonctions Lambda dans la console Amazon RDS. Lorsque vous configurez une instance de base de données RDS à utiliser avec Lambda, tenez compte des critères suivants :

- Pour se connecter à une base de données, votre fonction doit résider dans le même Amazon VPC où votre base de données s'exécute.
- Vous pouvez utiliser les bases de données Amazon RDS avec les moteurs MySQL, MariaDB, PostgreSQL ou Microsoft SQL Server.
- Vous pouvez également utiliser des clusters de base de données Aurora avec des moteurs MySQL ou PostgreSQL.
- Vous devez fournir un secret Secrets Manager pour l'authentification de la base de données.
- Un rôle IAM doit autoriser l'utilisation du secret et une stratégie d'approbation doit autoriser Amazon RDS à endosser le rôle.
- Le principal IAM qui utilise la console pour configurer la ressource Amazon RDS et la connecter à votre fonction doit disposer des autorisations suivantes :

Exemple de politique d'autorisations

> **Note**
>
> Vous n'avez besoin des autorisations de Proxy Amazon RDS que si vous configurez un Proxy Amazon RDS pour gérer un pool de connexions à votre base de données.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ec2:CreateSecurityGroup",
        "ec2:DescribeSecurityGroups",
        "ec2:DescribeSubnets",
        "ec2:DescribeVpcs",
        "ec2:AuthorizeSecurityGroupIngress",
        "ec2:AuthorizeSecurityGroupEgress",
        "ec2:RevokeSecurityGroupEgress",
        "ec2:CreateNetworkInterface",
        "ec2:DeleteNetworkInterface",
        "ec2:DescribeNetworkInterfaces"
```

```
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "rds-db:connect",
        "rds:CreateDBProxy",
        "rds:CreateDBInstance",
        "rds:CreateDBSubnetGroup",
        "rds:DescribeDBClusters",
        "rds:DescribeDBInstances",
        "rds:DescribeDBSubnetGroups",
        "rds:DescribeDBProxies",
        "rds:DescribeDBProxyTargets",
        "rds:DescribeDBProxyTargetGroups",
        "rds:RegisterDBProxyTargets",
        "rds:ModifyDBInstance",
        "rds:ModifyDBProxy"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "lambda:CreateFunction",
        "lambda:ListFunctions",
        "lambda:UpdateFunctionConfiguration"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:AttachRolePolicy",
        "iam:CreateRole",
        "iam:CreatePolicy"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "secretsmanager:GetResourcePolicy",
```

```
        "secretsmanager:GetSecretValue",
        "secretsmanager:DescribeSecret",
        "secretsmanager:ListSecretVersionIds",
        "secretsmanager:CreateSecret"
      ],
      "Resource": "*"
    }
  ]
}
```

Amazon RDS facture un tarif horaire pour les proxys en fonction de la taille de l'instance de base de données. Consultez la tarification des proxys RDS pour plus de détails. Pour plus d'informations sur les connexions de proxy en général, consultez Utilisation de Proxy Amazon RDS dans le Guide de l'utilisateur Amazon RDS.

Exigences SSL/TLS pour les connexions Amazon RDS

Pour établir des SSL/TLS connexions sécurisées à une instance de base de données Amazon RDS, votre fonction Lambda doit vérifier l'identité du serveur de base de données à l'aide d'un certificat sécurisé. Lambda gère ces certificats différemment selon le type de package de déploiement :

- Archives .zip : la gestion des certificats varie en fonction de l'environnement d'exécution :
  - Node.js 18 et versions antérieures : Lambda inclut automatiquement les certificats CA et RDS.
  - Node.js 20 et versions ultérieures : Lambda ne charge plus de certificats CA supplémentaires par défaut. Définissez la variable d'environnement `NODE_EXTRA_CA_CERTS` sur `/var/runtime/ca-cert.pem`.

  L'ajout de nouveaux certificats Amazon RDS aux environnements d'exécution gérés par Lambda peut Régions AWS prendre jusqu'à 4 semaines.
- Images de conteneur : les images AWS de base incluent uniquement les certificats CA. Si votre fonction se connecte à une instance de base de données Amazon RDS, vous devez inclure les certificats appropriés dans votre image de conteneur. Dans votre Dockerfile, téléchargez le bundle de certificats correspondant à l' Région AWS endroit où vous hébergez votre base de données. Exemple :

  ```
  RUN curl https://truststore.pki.rds.amazonaws.com/us-east-1/us-east-1-bundle.pem -o /us-east-1-bundle.pem
  ```

Cette commande télécharge le bundle de certificats Amazon RDS et l'enregistre sur le chemin absolu `/us-east-1-bundle.pem` dans le répertoire racine de votre conteneur. Lorsque vous configurez la connexion à la base de données dans le code de votre fonction, vous devez référencer ce chemin exact. Exemple :

Node.js

La fonction `readFileSync` est requise, car les clients de base de données Node.js ont besoin du contenu réel du certificat en mémoire, et pas seulement du chemin d'accès au fichier de certificat. Sans `readFileSync`, le client interprète la chaîne de chemin comme le contenu du certificat, ce qui entraîne une erreur « certificat autosigné dans la chaîne de certificats ».

Example Configuration de la connexion pour la fonction OCI Node.js

```
import { readFileSync } from 'fs';

// ...

let connectionConfig = {
    host: process.env.ProxyHostName,
    user: process.env.DBUserName,
    password: token,
    database: process.env.DBName,
    ssl: {
        ca: readFileSync('/us-east-1-bundle.pem') // Load RDS certificate content
 from file into memory
    }
};
```

Python

Example Configuration de la connexion pour la fonction OCI Python

```
connection = pymysql.connect(
    host=proxy_host_name,
    user=db_username,
    password=token,
    db=db_name,
    port=port,
    ssl={'ca': '/us-east-1-bundle.pem'}  #Path to the certificate in container
)
```

Java

Pour les fonctions Java utilisant des connexions JDBC, la chaîne de connexion doit inclure :

- `useSSL=true`
- `requireSSL=true`
- Paramètre `sslCA` indiquant l'emplacement du certificat Amazon RDS dans l'image de conteneur

Example Chaîne de connexion pour la fonction OCI Java

```
// Define connection string
String connectionString = String.format("jdbc:mysql://%s:%s/%s?
useSSL=true&requireSSL=true&sslCA=/us-east-1-bundle.pem", // Path to the certificate
 in container
        System.getenv("ProxyHostName"),
        System.getenv("Port"),
        System.getenv("DBName"));
```

.NET

Example Chaîne de connexion pour la connexion MySQL dans la fonction OCI .NET

```
/// Build the Connection String with the Token
string connectionString =
 $"Server={Environment.GetEnvironmentVariable("RDS_ENDPOINT")};" +
                         $"Port={Environment.GetEnvironmentVariable("RDS_PORT")};" +

 $"Uid={Environment.GetEnvironmentVariable("RDS_USERNAME")};" +
                         $"Pwd={authToken};" +
                         "SslMode=Required;" +
                         "SslCa=/us-east-1-bundle.pem";  // Path to the certificate
 in container
```

Go

Pour les fonctions Go utilisant des connexions MySQL, chargez le certificat Amazon RDS dans un groupe de certificats et enregistrez-le avec le pilote MySQL. La chaîne de connexion doit ensuite référencer cette configuration à l'aide du paramètre `tls`.

Example Code Go pour la connexion MySQL dans la fonction OCI

```
import (
    "crypto/tls"
    "crypto/x509"
    "os"
    "github.com/go-sql-driver/mysql"
)

...

// Create certificate pool and register TLS config
rootCertPool := x509.NewCertPool()
pem, err := os.ReadFile("/us-east-1-bundle.pem")  // Path to the certificate in
 container
if err != nil {
    panic("failed to read certificate file: " + err.Error())
}
if ok := rootCertPool.AppendCertsFromPEM(pem); !ok {
    panic("failed to append PEM")
}

mysql.RegisterTLSConfig("custom", &tls.Config{
    RootCAs: rootCertPool,
})

dsn := fmt.Sprintf("%s:%s@tcp(%s)/%s?allowCleartextPasswords=true&tls=custom",
    dbUser, authenticationToken, dbEndpoint, dbName,
)
```

Ruby

Example Configuration de connexion Ruby pour la fonction OCI

```
conn = Mysql2::Client.new(
    host: endpoint,
    username: user,
    password: token,
    port: port,
    database: db_name,
    sslca: '/us-east-1-bundle.pem',  # Path to the certificate in container
    sslverify: true
)
```

Connexion à une base de données Amazon RDS dans une fonction Lambda

L'exemple de code suivant illustre comment implémenter une fonction Lambda qui se connecte à une base de données Amazon RDS. La fonction effectue une simple requête de base de données et renvoie le résultat.

> **Note**
>
> Ces exemples de code ne sont valides que pour les packages de déploiement .zip. Si vous déployez votre fonction à l'aide d'une image de conteneur, vous devez spécifier le fichier de certificat Amazon RDS dans le code de votre fonction, comme expliqué dans la section précédente.

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus sur GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Connexion à une base de données Amazon RDS dans une fonction Lambda à l'aide de .NET.

```
using System.Data;
using System.Text.Json;
using Amazon.Lambda.APIGatewayEvents;
using Amazon.Lambda.Core;
using MySql.Data.MySqlClient;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace aws_rds;
```

```
public class InputModel
{
    public string key1 { get; set; }
    public string key2 { get; set; }
}

public class Function
{
    /// <summary>
    // Handles the Lambda function execution for connecting to RDS using IAM
 authentication.
    /// </summary>
    /// <param name="input">The input event data passed to the Lambda function</
param>
    /// <param name="context">The Lambda execution context that provides runtime
 information</param>
    /// <returns>A response object containing the execution result</returns>

    public async Task<APIGatewayProxyResponse>
 FunctionHandler(APIGatewayProxyRequest request, ILambdaContext context)
    {
        // Sample Input: {"body": "{\"key1\":\"20\", \"key2\":\"25\"}"}
        var input = JsonSerializer.Deserialize<InputModel>(request.Body);

        /// Obtain authentication token
        var authToken = RDSAuthTokenGenerator.GenerateAuthToken(
            Environment.GetEnvironmentVariable("RDS_ENDPOINT"),
            Convert.ToInt32(Environment.GetEnvironmentVariable("RDS_PORT")),
            Environment.GetEnvironmentVariable("RDS_USERNAME")
        );

        /// Build the Connection String with the Token
        string connectionString =
 $"Server={Environment.GetEnvironmentVariable("RDS_ENDPOINT")};" +

 $"Port={Environment.GetEnvironmentVariable("RDS_PORT")};" +

 $"Uid={Environment.GetEnvironmentVariable("RDS_USERNAME")};" +
                                  $"Pwd={authToken};";


        try
        {
            await using var connection = new MySqlConnection(connectionString);
```

```
            await connection.OpenAsync();

            const string sql = "SELECT @param1 + @param2 AS Sum";

            await using var command = new MySqlCommand(sql, connection);
            command.Parameters.AddWithValue("@param1", int.Parse(input.key1 ??
 "0"));
            command.Parameters.AddWithValue("@param2", int.Parse(input.key2 ??
 "0"));

            await using var reader = await command.ExecuteReaderAsync();
            if (await reader.ReadAsync())
            {
                int result = reader.GetInt32("Sum");

                //Sample Response: {"statusCode":200,"body":"{\"message\":\"The
 sum is: 45\"}","isBase64Encoded":false}
                return new APIGatewayProxyResponse
                {
                    StatusCode = 200,
                    Body = JsonSerializer.Serialize(new { message = $"The sum is:
 {result}" })
                };
            }

        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }

        return new APIGatewayProxyResponse
        {
            StatusCode = 500,
            Body = JsonSerializer.Serialize(new { error = "Internal server
 error" })
        };
    }
}
```

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus sur GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Connexion à une base de données Amazon RDS dans une fonction Lambda à l'aide de Go.

```
/*
Golang v2 code here.
*/

package main

import (
 "context"
 "database/sql"
 "encoding/json"
 "fmt"
 "os"

 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/rds/auth"
 _ "github.com/go-sql-driver/mysql"
)

type MyEvent struct {
 Name string `json:"name"`
}

func HandleRequest(event *MyEvent) (map[string]interface{}, error) {

 var dbName string = os.Getenv("DatabaseName")
 var dbUser string = os.Getenv("DatabaseUser")
 var dbHost string = os.Getenv("DBHost") // Add hostname without https
 var dbPort int = os.Getenv("Port")      // Add port number
 var dbEndpoint string = fmt.Sprintf("%s:%d", dbHost, dbPort)
 var region string = os.Getenv("AWS_REGION")
```

```
 cfg, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  panic("configuration error: " + err.Error())
 }

 authenticationToken, err := auth.BuildAuthToken(
  context.TODO(), dbEndpoint, region, dbUser, cfg.Credentials)
 if err != nil {
  panic("failed to create authentication token: " + err.Error())
 }

 dsn := fmt.Sprintf("%s:%s@tcp(%s)/%s?tls=true&allowCleartextPasswords=true",
  dbUser, authenticationToken, dbEndpoint, dbName,
 )

 db, err := sql.Open("mysql", dsn)
 if err != nil {
  panic(err)
 }

 defer db.Close()

 var sum int
 err = db.QueryRow("SELECT ?+? AS sum", 3, 2).Scan(&sum)
 if err != nil {
  panic(err)
 }
 s := fmt.Sprint(sum)
 message := fmt.Sprintf("The selected sum is: %s", s)

 messageBytes, err := json.Marshal(message)
 if err != nil {
  return nil, err
 }

 messageString := string(messageBytes)
 return map[string]interface{}{
  "statusCode": 200,
  "headers":    map[string]string{"Content-Type": "application/json"},
  "body":       messageString,
 }, nil
}
```

```
func main() {
 lambda.Start(HandleRequest)
}
```

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus sur GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Connexion à une base de données Amazon RDS dans une fonction Lambda à l'aide de Java.

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.rdsdata.RdsDataClient;
import software.amazon.awssdk.services.rdsdata.model.ExecuteStatementRequest;
import software.amazon.awssdk.services.rdsdata.model.ExecuteStatementResponse;
import software.amazon.awssdk.services.rdsdata.model.Field;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

public class RdsLambdaHandler implements
 RequestHandler<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent> {

    @Override
    public APIGatewayProxyResponseEvent handleRequest(APIGatewayProxyRequestEvent
 event, Context context) {
        APIGatewayProxyResponseEvent response = new
 APIGatewayProxyResponseEvent();
```

```
        try {
            // Obtain auth token
            String token = createAuthToken();

            // Define connection configuration
            String connectionString = String.format("jdbc:mysql://%s:%s/%s?
useSSL=true&requireSSL=true",
                    System.getenv("ProxyHostName"),
                    System.getenv("Port"),
                    System.getenv("DBName"));

            // Establish a connection to the database
            try (Connection connection =
 DriverManager.getConnection(connectionString, System.getenv("DBUserName"),
 token);
                 PreparedStatement statement =
 connection.prepareStatement("SELECT ? + ? AS sum")) {

                statement.setInt(1, 3);
                statement.setInt(2, 2);

                try (ResultSet resultSet = statement.executeQuery()) {
                    if (resultSet.next()) {
                        int sum = resultSet.getInt("sum");
                        response.setStatusCode(200);
                        response.setBody("The selected sum is: " + sum);
                    }
                }
            }

        } catch (Exception e) {
            response.setStatusCode(500);
            response.setBody("Error: " + e.getMessage());
        }

        return response;
    }

    private String createAuthToken() {
        // Create RDS Data Service client
        RdsDataClient rdsDataClient = RdsDataClient.builder()
                .region(Region.of(System.getenv("AWS_REGION")))
                .credentialsProvider(DefaultCredentialsProvider.create())
                .build();
```

```
        // Define authentication request
        ExecuteStatementRequest request = ExecuteStatementRequest.builder()
                .resourceArn(System.getenv("ProxyHostName"))
                .secretArn(System.getenv("DBUserName"))
                .database(System.getenv("DBName"))
                .sql("SELECT 'RDS IAM Authentication'")
                .build();

        // Execute request and obtain authentication token
        ExecuteStatementResponse response =
 rdsDataClient.executeStatement(request);
        Field tokenField = response.records().get(0).get(0);

        return tokenField.stringValue();
    }
}
```

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus sur GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Connexion à une base de données Amazon RDS dans une fonction Lambda à l'aide de. JavaScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
/*
Node.js code here.
*/
// ES6+ example
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';
```

```
async function createAuthToken() {
  // Define connection authentication parameters
  const dbinfo = {

    hostname: process.env.ProxyHostName,
    port: process.env.Port,
    username: process.env.DBUserName,
    region: process.env.AWS_REGION,

  }

  // Create RDS Signer object
  const signer = new Signer(dbinfo);

  // Request authorization token from RDS, specifying the username
  const token = await signer.getAuthToken();
  return token;
}

async function dbOps() {

  // Obtain auth token
  const token = await createAuthToken();
  // Define connection configuration
  let connectionConfig = {
    host: process.env.ProxyHostName,
    user: process.env.DBUserName,
    password: token,
    database: process.env.DBName,
    ssl: 'Amazon RDS'
  }
  // Create the connection to the DB
  const conn = await mysql.createConnection(connectionConfig);
  // Obtain the result of the query
  const [res,] = await conn.execute('select ?+? as sum', [3, 2]);
  return res;

}

export const handler = async (event) => {
  // Execute database flow
  const result = await dbOps();
  // Return result
  return {
```

```
    statusCode: 200,
    body: JSON.stringify("The selected sum is: " + result[0].sum)
  }
};
```

Connexion à une base de données Amazon RDS dans une fonction Lambda à l'aide de. TypeScript

```
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

// RDS settings
// Using '!' (non-null assertion operator) to tell the TypeScript compiler that
 the DB settings are not null or undefined,
const proxy_host_name = process.env.PROXY_HOST_NAME!
const port = parseInt(process.env.PORT!)
const db_name = process.env.DB_NAME!
const db_user_name = process.env.DB_USER_NAME!
const aws_region = process.env.AWS_REGION!


async function createAuthToken(): Promise<string> {

    // Create RDS Signer object
    const signer = new Signer({
        hostname: proxy_host_name,
        port: port,
        region: aws_region,
        username: db_user_name
    });

    // Request authorization token from RDS, specifying the username
    const token = await signer.getAuthToken();
    return token;
}

async function dbOps(): Promise<mysql.QueryResult | undefined> {
    try {
        // Obtain auth token
        const token = await createAuthToken();
        const conn = await mysql.createConnection({
```

```
            host: proxy_host_name,
            user: db_user_name,
            password: token,
            database: db_name,
            ssl: 'Amazon RDS' // Ensure you have the CA bundle for SSL connection
        });
        const [rows, fields] = await conn.execute('SELECT ? + ? AS sum', [3, 2]);
        console.log('result:', rows);
        return rows;
    }
    catch (err) {
        console.log(err);
    }
}

export const lambdaHandler = async (event: any): Promise<{ statusCode: number;
 body: string }> => {
    // Execute database flow
    const result = await dbOps();

    // Return error is result is undefined
    if (result == undefined)
        return {
            statusCode: 500,
            body: JSON.stringify(`Error with connection to DB host`)
        }

    // Return result
    return {
        statusCode: 200,
        body: JSON.stringify(`The selected sum is: ${result[0].sum}`)
    };
};
```

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus sur GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Connexion à une base de données Amazon RDS dans une fonction Lambda à l'aide de PHP.

```
<?php
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;
use Aws\Rds\AuthTokenGenerator;
use Aws\Credentials\CredentialProvider;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }


    private function getAuthToken(): string {
        // Define connection authentication parameters
        $dbConnection = [
            'hostname' => getenv('DB_HOSTNAME'),
            'port' => getenv('DB_PORT'),
            'username' => getenv('DB_USERNAME'),
            'region' => getenv('AWS_REGION'),
        ];
```

```
        // Create RDS AuthTokenGenerator object
        $generator = new
AuthTokenGenerator(CredentialProvider::defaultProvider());

        // Request authorization token from RDS, specifying the username
        return $generator->createToken(
            $dbConnection['hostname'] . ':' . $dbConnection['port'],
            $dbConnection['region'],
            $dbConnection['username']
        );
    }

    private function getQueryResults() {
        // Obtain auth token
        $token = $this->getAuthToken();

        // Define connection configuration
        $connectionConfig = [
            'host' => getenv('DB_HOSTNAME'),
            'user' => getenv('DB_USERNAME'),
            'password' => $token,
            'database' => getenv('DB_NAME'),
        ];

        // Create the connection to the DB
        $conn = new PDO(

"mysql:host={$connectionConfig['host']};dbname={$connectionConfig['database']}",
            $connectionConfig['user'],
            $connectionConfig['password'],
            [
                PDO::MYSQL_ATTR_SSL_CA => '/path/to/rds-ca-2019-root.pem',
                PDO::MYSQL_ATTR_SSL_VERIFY_SERVER_CERT => true,
            ]
        );

        // Obtain the result of the query
        $stmt = $conn->prepare('SELECT ?+? AS sum');
        $stmt->execute([3, 2]);

        return $stmt->fetch(PDO::FETCH_ASSOC);
    }
```

```
    /**
     * @param mixed $event
     * @param Context $context
     * @return array
     */
    public function handle(mixed $event, Context $context): array
    {
        $this->logger->info("Processing query");

        // Execute database flow
        $result = $this->getQueryResults();

        return [
            'sum' => $result['sum']
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus sur GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Connexion à une base de données Amazon RDS dans une fonction Lambda à l'aide de Python.

```
import json
import os
import boto3
import pymysql

# RDS settings
proxy_host_name = os.environ['PROXY_HOST_NAME']
```

```
port = int(os.environ['PORT'])
db_name = os.environ['DB_NAME']
db_user_name = os.environ['DB_USER_NAME']
aws_region = os.environ['AWS_REGION']


# Fetch RDS Auth Token
def get_auth_token():
    client = boto3.client('rds')
    token = client.generate_db_auth_token(
        DBHostname=proxy_host_name,
        Port=port
        DBUsername=db_user_name
        Region=aws_region
    )
    return token

def lambda_handler(event, context):
    token = get_auth_token()
    try:
        connection = pymysql.connect(
            host=proxy_host_name,
            user=db_user_name,
            password=token,
            db=db_name,
            port=port,
            ssl={'ca': 'Amazon RDS'}  # Ensure you have the CA bundle for SSL
 connection
        )

        with connection.cursor() as cursor:
            cursor.execute('SELECT %s + %s AS sum', (3, 2))
            result = cursor.fetchone()

        return result

    except Exception as e:
        return (f"Error: {str(e)}")  # Return an error message if an exception
 occurs
```

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus sur GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Connexion à une base de données Amazon RDS dans une fonction Lambda à l'aide de Ruby.

```
# Ruby code here.

require 'aws-sdk-rds'
require 'json'
require 'mysql2'

def lambda_handler(event:, context:)
  endpoint = ENV['DBEndpoint'] # Add the endpoint without https"
  port = ENV['Port']           # 3306
  user = ENV['DBUser']
  region = ENV['DBRegion']     # 'us-east-1'
  db_name = ENV['DBName']

  credentials = Aws::Credentials.new(
    ENV['AWS_ACCESS_KEY_ID'],
    ENV['AWS_SECRET_ACCESS_KEY'],
    ENV['AWS_SESSION_TOKEN']
  )
  rds_client = Aws::RDS::AuthTokenGenerator.new(
    region: region,
    credentials: credentials
  )

  token = rds_client.auth_token(
    endpoint: endpoint+ ':' + port,
    user_name: user,
    region: region
  )

  begin
    conn = Mysql2::Client.new(
```

```
      host: endpoint,
      username: user,
      password: token,
      port: port,
      database: db_name,
      sslca: '/var/task/global-bundle.pem',
      sslverify: true,
      enable_cleartext_plugin: true
    )
    a = 3
    b = 2
    result = conn.query("SELECT #{a} + #{b} AS sum").first['sum']
    puts result
    conn.close
    {
      statusCode: 200,
      body: result.to_json
    }
  rescue => e
    puts "Database connection failed due to #{e}"
  end
end
```

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus sur GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Connexion à une base de données Amazon RDS dans une fonction Lambda à l'aide de Rust.

```
use aws_config::BehaviorVersion;
use aws_credential_types::provider::ProvideCredentials;
use aws_sigv4::{
    http_request::{sign, SignableBody, SignableRequest, SigningSettings},
    sign::v4,
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
```

```
use serde_json::{json, Value};
use sqlx::postgres::PgConnectOptions;
use std::env;
use std::time::{Duration, SystemTime};

const RDS_CERTS: &[u8] = include_bytes!("global-bundle.pem");

async fn generate_rds_iam_token(
    db_hostname: &str,
    port: u16,
    db_username: &str,
) -> Result<String, Error> {
    let config = aws_config::load_defaults(BehaviorVersion::v2024_03_28()).await;

    let credentials = config
        .credentials_provider()
        .expect("no credentials provider found")
        .provide_credentials()
        .await
        .expect("unable to load credentials");
    let identity = credentials.into();
    let region = config.region().unwrap().to_string();

    let mut signing_settings = SigningSettings::default();
    signing_settings.expires_in = Some(Duration::from_secs(900));
    signing_settings.signature_location =
 aws_sigv4::http_request::SignatureLocation::QueryParams;

    let signing_params = v4::SigningParams::builder()
        .identity(&identity)
        .region(&region)
        .name("rds-db")
        .time(SystemTime::now())
        .settings(signing_settings)
        .build()?;

    let url = format!(
        "https://{db_hostname}:{port}/?Action=connect&DBUser={db_user}",
        db_hostname = db_hostname,
        port = port,
        db_user = db_username
    );

    let signable_request =
```

```
        SignableRequest::new("GET", &url, std::iter::empty(),
 SignableBody::Bytes(&[]))
            .expect("signable request");

    let (signing_instructions, _signature) =
        sign(signable_request, &signing_params.into())?.into_parts();

    let mut url = url::Url::parse(&url).unwrap();
    for (name, value) in signing_instructions.params() {
        url.query_pairs_mut().append_pair(name, &value);
    }

    let response = url.to_string().split_off("https://".len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(handler)).await
}

async fn handler(_event: LambdaEvent<Value>) -> Result<Value, Error> {
    let db_host = env::var("DB_HOSTNAME").expect("DB_HOSTNAME must be set");
    let db_port = env::var("DB_PORT")
        .expect("DB_PORT must be set")
        .parse::<u16>()
        .expect("PORT must be a valid number");
    let db_name = env::var("DB_NAME").expect("DB_NAME must be set");
    let db_user_name = env::var("DB_USERNAME").expect("DB_USERNAME must be set");

    let token = generate_rds_iam_token(&db_host, db_port, &db_user_name).await?;

    let opts = PgConnectOptions::new()
        .host(&db_host)
        .port(db_port)
        .username(&db_user_name)
        .password(&token)
        .database(&db_name)
        .ssl_root_cert_from_pem(RDS_CERTS.to_vec())
        .ssl_mode(sqlx::postgres::PgSslMode::Require);

    let pool = sqlx::postgres::PgPoolOptions::new()
        .connect_with(opts)
```

```
        .await?;

    let result: i32 = sqlx::query_scalar("SELECT $1 + $2")
        .bind(3)
        .bind(2)
        .fetch_one(&pool)
        .await?;

    println!("Result: {:?}", result);

    Ok(json!({
        "statusCode": 200,
        "content-type": "text/plain",
        "body": format!("The selected sum is: {result}")
    }))
}
```

Traitement des notifications d'événements provenant d'Amazon RDS

Vous pouvez utiliser Lambda pour traiter les notifications d'événements d'une base de données Amazon RDS. Amazon RDS envoie des notifications à une rubrique Amazon Simple Notification Service (Amazon SNS) que vous pouvez configurer pour invoquer une fonction Lambda. Amazon SNS enveloppe le message d'Amazon RDS dans son propre document d'événement, et l'envoie à votre fonction.

Pour plus d'informations sur la configuration d'une base de données Amazon RDS pour envoyer des notifications, consultez Utilisation des notifications d'événements Amazon RDS.

Example Message Amazon RDS dans un événement Amazon SNS

```
{
        "Records": [
          {
            "EventVersion": "1.0",
            "EventSubscriptionArn": "arn:aws:sns:us-east-2:123456789012:rds-
lambda:21be56ed-a058-49f5-8c98-aedd2564c486",
            "EventSource": "aws:sns",
            "Sns": {
              "SignatureVersion": "1",
              "Timestamp": "2023-01-02T12:45:07.000Z",
```

```
              "Signature": "tcc6faL2yUC6dgZdmrwh1Y4cGa/ebXEkAi6RibDsvpi
+tE/1+82j...65r==",
              "SigningCertUrl": "https://sns.us-east-2.amazonaws.com/
SimpleNotificationService-ac565b8b1a6c5d002d285f9598aa1d9b.pem",
              "MessageId": "95df01b4-ee98-5cb9-9903-4c221d41eb5e",
              "Message": "{\"Event Source\":\"db-instance\",\"Event Time\":\"2023-01-02
 12:45:06.000\",\"Identifier Link\":\"https://console.aws.amazon.com/rds/home?
region=eu-west-1#dbinstance:id=dbinstanceid\",\"Source ID\":\"dbinstanceid\",\"Event ID
\":\"http://docs.amazonwebservices.com/AmazonRDS/latest/UserGuide/USER_Events.html#RDS-
EVENT-0002\",\"Event Message\":\"Finished DB Instance backup\"}",
              "MessageAttributes": {},
              "Type": "Notification",
              "UnsubscribeUrl": "https://sns.us-east-2.amazonaws.com/?
Action=Unsubscribe&amp;SubscriptionArn=arn:aws:sns:us-east-2:123456789012:test-
lambda:21be56ed-a058-49f5-8c98-aedd2564c486",
              "TopicArn":"arn:aws:sns:us-east-2:123456789012:sns-lambda",
              "Subject": "RDS Notification Message"
            }
          }
        ]
      }
```

Tutoriel complet Lambda et Amazon RDS

- Utilisation d'une fonction Lambda pour accéder à une base de données Amazon RDS — Dans le Guide de l'utilisateur Amazon RDS, découvrez comment utiliser une fonction Lambda pour écrire des données dans une base de données Amazon RDS via Proxy Amazon RDS. Votre fonction Lambda lira les enregistrements d'une file d'attente Amazon SQS et écrira de nouveaux éléments dans une table de votre base de données chaque fois qu'un message sera ajouté.

Sélectionner un service de base de données pour vos applications basées sur Lambda

De nombreuses applications sans serveur ont besoin de stocker et de récupérer des données. AWS propose plusieurs options de base de données qui fonctionnent avec les fonctions Lambda. Les deux options les plus populaires sont Amazon DynamoDB, un service de base de données NoSQL, et Amazon RDS, une solution de base de données relationnelle traditionnelle. Les sections suivantes expliquent les principales différences entre ces services lorsque vous les utilisez avec Lambda, et vous aident à sélectionner le service de base de données adapté à votre application sans serveur.

Pour en savoir plus sur les autres services de base de données proposés par AWS, et pour comprendre leurs cas d'utilisation et leurs inconvénients de manière plus générale, voir Choisir un service de AWS base de données. Tous les services de base de données AWS sont compatibles avec Lambda, mais ils ne sont pas nécessairement tous adaptés à votre cas d'utilisation particulier.

Quelles sont les options proposées lors de la sélection d'un service de base de données avec Lambda ?

AWS propose plusieurs services de base de données. Pour les applications sans serveur, DynamoDB et Amazon RDS sont deux des options les plus populaires.

- DynamoDB est un service de base de données NoSQL entièrement géré, optimisé pour les applications sans serveur. Il offre une mise à l'échelle transparente et des performances constantes de l'ordre de quelques millisecondes à n'importe quelle échelle.
- Amazon RDS est un service de base de données relationnelle géré qui prend en charge plusieurs moteurs de base de données, dont MySQL et PostgreSQL. Il fournit les fonctionnalités familières de SQL avec une infrastructure gérée.

Recommandations si vous connaissez déjà vos besoins

Si vous connaissez déjà bien vos besoins, voici nos recommandations de base :

Nous recommandons DynamoDB pour les applications sans serveur qui ont besoin de performances constantes à faible latence et d'une mise à l'échelle automatique, et qui ne nécessitent pas de jointures ou de transactions complexes. Cette solution est particulièrement adaptée aux applications basées sur Lambda en raison de sa nature sans serveur.

Amazon RDS est un meilleur choix lorsque vous avez besoin de requêtes SQL complexes, de jointures ou si vous avez des applications existantes utilisant des bases de données relationnelles. Sachez toutefois que la connexion des fonctions Lambda à Amazon RDS nécessite une configuration supplémentaire et peut avoir un impact sur les temps de démarrage à froid.

Éléments à prendre en compte lors de la sélection d'un service de base de données

Lorsque vous choisissez entre DynamoDB et Amazon RDS pour vos applications Lambda, tenez compte des facteurs suivants :

- Gestion des connexions et démarrages à froid

- Modèles d’accès aux données
- Complexité des requêtes
- Exigences de cohérence des données
- Caractéristiques de mise à l’échelle
- Modèle de coût

La compréhension de ces facteurs vous permettra de choisir l’option qui répond le mieux aux besoins de votre cas d’utilisation.

Gestion des connexions et démarrages à froid

- DynamoDB utilise une API HTTP pour toutes les opérations. Les fonctions Lambda peuvent effectuer des requêtes immédiatement sans maintenir de connexion, ce qui améliore les performances de démarrage à froid. Chaque demande est authentifiée à l'aide AWS d'informations d'identification sans surcoût de connexion.
- Amazon RDS nécessite de gérer des groupes de connexions, car il utilise des connexions de base de données traditionnelles. Cela peut affecter les démarrages à froid, car les nouvelles instances Lambda doivent établir des connexions. Vous devrez mettre en œuvre des stratégies de regroupement de connexions et éventuellement utiliser le Proxy Amazon RDS pour gérer efficacement les connexions. Notez que l’utilisation du Proxy Amazon RDS entraîne des coûts supplémentaires.

Modèles d’accès aux données

- DynamoDB fonctionne mieux avec les modèles d’accès connus et les modèles à table unique. Il est idéal pour les applications Lambda qui ont besoin d’un accès constant à faible latence aux données sur la base de clés primaires ou d’index secondaires.
- Amazon RDS offre de la flexibilité pour les requêtes complexes et les modèles d’accès changeants. Il est mieux adapté lorsque vos fonctions Lambda doivent effectuer des requêtes uniques et personnalisées, ou des jointures complexes sur plusieurs tables.

Complexité des requêtes

- DynamoDB excelle dans les opérations simples basées sur les clés et dans les modèles d’accès prédéfinis. Les requêtes complexes doivent être conçues autour de structures d’index, et les jointures doivent être gérées dans le code de l’application.

- Amazon RDS prend en charge les requêtes SQL complexes avec des jointures, des sous-requêtes et des agrégations. Cela peut simplifier le code de votre fonction Lambda lorsque des opérations de données complexes sont nécessaires.

Exigences de cohérence des données

- DynamoDB propose à la fois des options de cohérence finale et de cohérence forte, la cohérence forte étant disponible pour les lectures d'un seul élément. Les transactions sont prises en charge, mais avec certaines restrictions.
- Amazon RDS offre une conformité totale en matière d'atomicité, de cohérence, d'isolation et de durabilité (ACID) et une prise en charge des transactions complexes. Si vos fonctions Lambda nécessitent des transactions complexes ou une cohérence forte entre plusieurs enregistrements, Amazon RDS pourrait être plus adapté.

Caractéristiques de mise à l'échelle

- DynamoDB s'adapte automatiquement à votre charge de travail. Il peut gérer les pics de trafic soudains provenant des fonctions Lambda sans pré-approvisionnement. Vous pouvez utiliser le mode de capacité à la demande pour ne payer que ce que vous utilisez, ce qui correspond parfaitement au modèle de mise à l'échelle de Lambda.
- Amazon RDS dispose d'une capacité fixe en fonction de la taille d'instance que vous choisissez. Si plusieurs fonctions Lambda tentent de se connecter simultanément, vous risquez de dépasser votre quota de connexions. Vous devez gérer avec soin les groupes de connexions et éventuellement implémenter une logique de nouvelle tentative.

Modèle de coût

- La tarification de DynamoDB est parfaitement adaptée aux applications sans serveur. Avec une capacité à la demande, vous ne payez que pour les lectures et écritures réellement utilisées par vos fonctions Lambda. Le temps d'inactivité n'est pas facturé.
- Amazon RDS facture l'instance en cours d'exécution, quelle que soit son utilisation. Cela peut être moins rentable pour les charges de travail sporadiques, qui sont courantes dans les applications sans serveur. Toutefois, cela peut s'avérer plus économique pour les charges de travail à haut débit avec une utilisation constante.

Commencer avec le service de base de données que vous avez choisi

Maintenant que vous avez pris connaissance des critères de sélection entre DynamoDB et Amazon RDS et des principales différences entre les deux, vous pouvez sélectionner l'option qui répond le mieux à vos besoins et utiliser les ressources suivantes pour commencer à l'utiliser.

DynamoDB

Mise en route avec DynamoDB grâce aux ressources suivantes

- Pour une présentation du service DynamoDB, consultez Qu'est-ce que DynamoDB ? dans le Guide du développeur Amazon DynamoDB
- Suivez le didacticiel Utilisation de Lambda avec API Gateway pour découvrir un exemple d'utilisation d'une fonction Lambda pour effectuer des opérations CRUD sur une table DynamoDB en réponse à une demande d'API.
- Lisez Programming with DynamoDB et AWS SDKs le manuel du développeur Amazon DynamoDB pour en savoir plus sur la façon d'accéder à DynamoDB depuis votre fonction Lambda en utilisant l'un des. AWS SDKs

Amazon RDS

Mise en route avec Amazon RDS grâce aux ressources suivantes

- Pour une présentation du service Amazon RDS, consultez l'article Qu'est-ce que Amazon Relational Database Service (Amazon RDS) ? dans le Guide de l'utilisateur Amazon Relational Database Service.
- Suivez le didacticiel Utilisation d'une fonction Lambda pour accéder à une base de données Amazon RDS dans le Guide de l'utilisateur Amazon Relational Database Service.
- Pour en savoir plus sur l'utilisation de Lambda avec Amazon RDS, consultez the section called "RDS".

Traiter les notifications d'événements Amazon S3 avec Lambda

Vous pouvez utiliser Lambda pour traiter les notifications d'événement d'Amazon Simple Storage Service. Amazon S3 peut envoyer un événement à une fonction Lambda lors de la création ou de la suppression d'un objet. Vous configurez des paramètres de notification sur un compartiment et accordez à Amazon S3 l'autorisation d'appeler une fonction sur la stratégie d'autorisations basée sur une ressource de la fonction.

> **Warning**
>
> Si votre fonction Lambda utilise le même compartiment que celui qui la déclenche, la fonction risque de s'exécuter en boucle. Par exemple, si le compartiment déclenche une fonction chaque fois qu'un objet est chargé et que la fonction charge un objet dans le compartiment, la fonction se déclenche elle-même indirectement. Afin d'éviter cela, utilisez deux compartiments ou configurez le déclencheur pour qu'il s'applique uniquement à un préfixe utilisé pour les objets entrants.

Amazon S3 appelle votre fonction de manière asynchrone avec un événement contenant des détails sur l'objet. L'exemple suivant montre un événement envoyé par Amazon S3 lors du chargement d'un package de déploiement vers Amazon S3.

Example Evénement de notification Amazon S3

```
{
  "Records": [
    {
      "eventVersion": "2.1",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-2",
      "eventTime": "2019-09-03T19:37:27.192Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "AWS:AIDAINPONIXQXHT3IKHL2"
      },
      "requestParameters": {
        "sourceIPAddress": "205.255.255.255"
      },
      "responseElements": {
        "x-amz-request-id": "D82B88E5F771F645",
```

```
        "x-amz-id-2":
 "vlR7PnpV2Ce81l0PRw6jlUpck7Jo5ZsQjryTjKlc5aLWGVHPZLj5NeC6qMa0emYBDXOo6QBU0Wo="
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "828aa6fc-f7b5-4305-8584-487c791949c1",
        "bucket": {
          "name": "amzn-s3-demo-bucket",
          "ownerIdentity": {
            "principalId": "A3I5XTEXAMAI3E"
          },
          "arn": "arn:aws:s3:::lambda-artifacts-deafc19498e3f2df"
        },
        "object": {
          "key": "b21b84d653bb07b05b1e6b33684dc11b",
          "size": 1305107,
          "eTag": "b21b84d653bb07b05b1e6b33684dc11b",
          "sequencer": "0C0F6F405D6ED209E1"
        }
      }
    }
  ]
}
```

Pour appeler votre fonction, Amazon S3 a besoin d'une autorisation de la stratégie basée sur une ressource. Lorsque vous configurez un déclencheur Amazon S3 dans la console Lambda, cette dernière modifie la stratégie basée sur une ressource pour permettre à Amazon S3 d'appeler la fonction si le nom du compartiment et l'ID de compte correspondent. Si vous configurez la notification dans Amazon S3, vous utilisez l'API Lambda pour mettre à jour la stratégie. Vous pouvez également utiliser l'API Lambda pour accorder une autorisation à un autre compte ou limiter l'autorisation à un alias désigné.

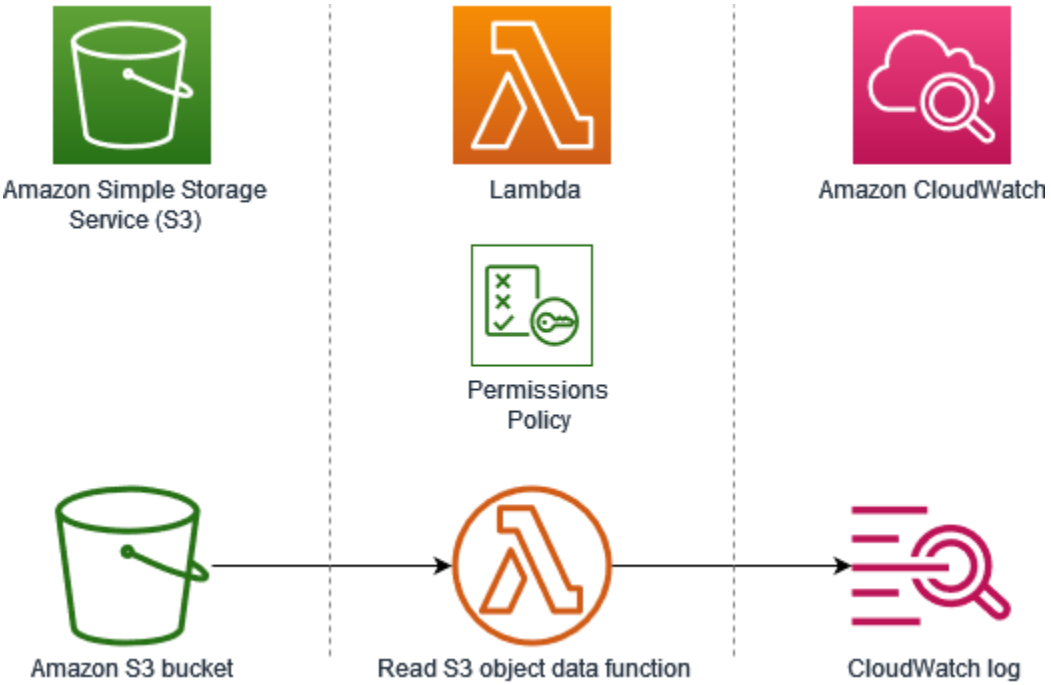
Si votre fonction utilise le AWS SDK pour gérer les ressources Amazon S3, elle a également besoin des autorisations Amazon S3 dans son rôle d'exécution.

Rubriques

- Didacticiel : utilisation d'un déclencheur Amazon S3 pour invoquer une fonction Lambda
- Didacticiel : Utilisation d'un déclencheur Amazon S3 pour créer des images miniatures

Didacticiel : utilisation d'un déclencheur Amazon S3 pour invoquer une fonction Lambda

Dans ce didacticiel, vous allez utiliser la console pour créer une fonction Lambda et configurer un déclencheur pour un compartiment Amazon Simple Storage Service (Amazon S3). Chaque fois que vous ajoutez un objet à votre compartiment Amazon S3, votre fonction s'exécute et affiche le type d'objet dans Amazon CloudWatch Logs.

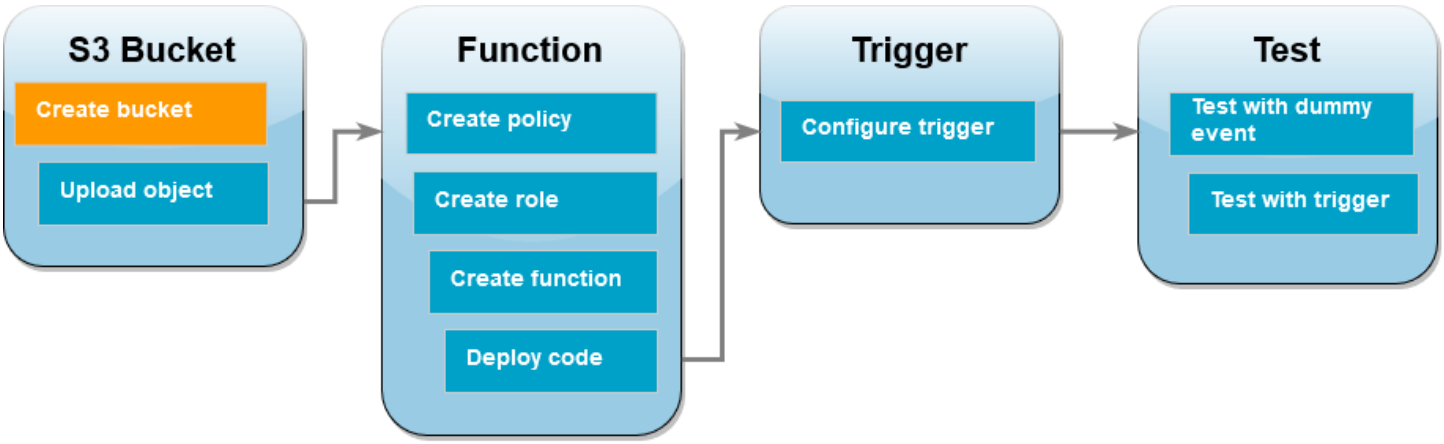


Ce tutoriel montre comment :

1. Créez un compartiment Amazon S3.
2. Créez une fonction Lambda qui renvoie le type d'objet des objets dans un compartiment Amazon S3.
3. Configurez un déclencheur Lambda qui invoque votre fonction lorsque des objets sont chargés dans votre compartiment.
4. Testez votre fonction, d'abord avec un événement fictif, puis en utilisant le déclencheur.

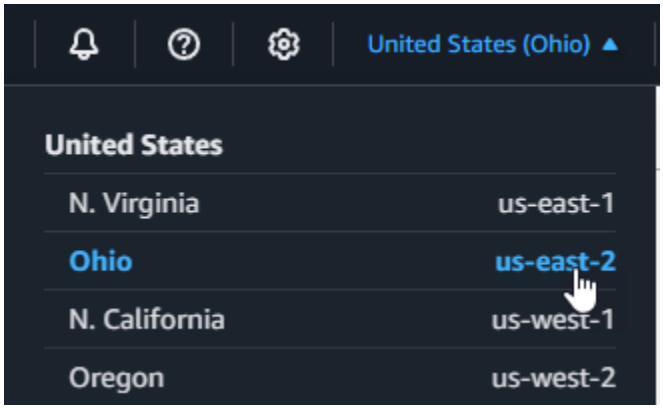
En suivant ces étapes, vous apprendrez à configurer une fonction Lambda pour qu'elle s'exécute chaque fois que des objets sont ajoutés ou supprimés d'un compartiment Amazon S3. Vous pouvez compléter ce didacticiel en n'utilisant que la AWS Management Console.

Créer un compartiment Amazon S3



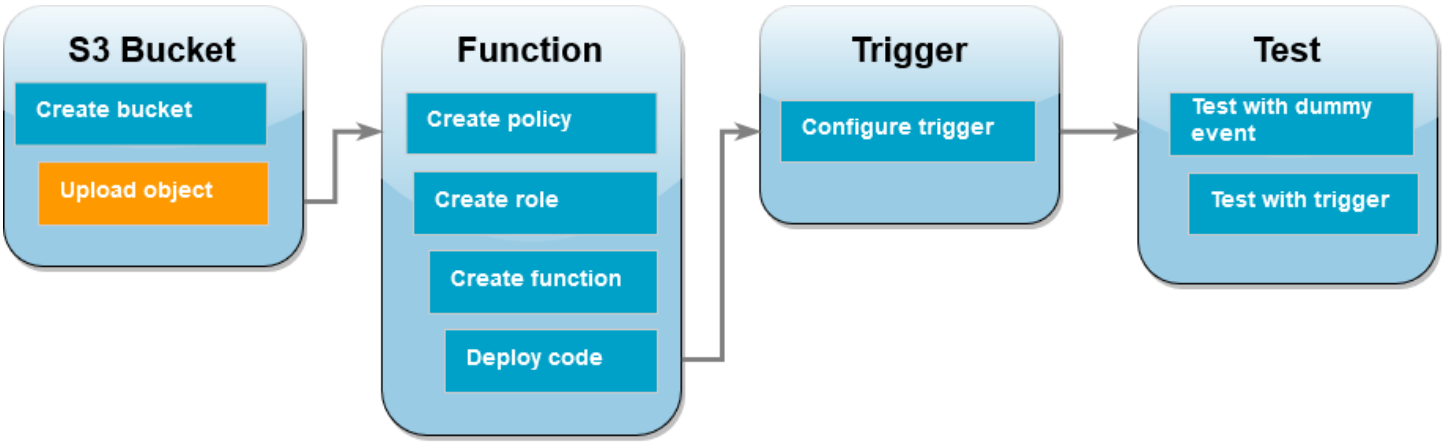
Pour créer un compartiment Amazon S3

1. Ouvrez la console Amazon S3 et sélectionnez la page Compartiments à usage général.
2. Sélectionnez le Région AWS plus proche de votre situation géographique. Vous pouvez modifier votre région à l'aide de la liste déroulante en haut de l'écran. Plus loin dans le didacticiel, vous devez créer votre fonction Lambda dans la même région.



3. Choisissez Create bucket (Créer un compartiment).
4. Sous Configuration générale, procédez comme suit :
   a. Pour le type de compartiment, veillez à sélectionner Usage général.
   b. Pour le nom du compartiment, saisissez un nom unique au monde qui respecte les règles de dénomination du compartiment Amazon S3. Les noms de compartiment peuvent contenir uniquement des lettres minuscules, des chiffres, de points (.) et des traits d'union (-).
5. Conservez les valeurs par défaut de toutes les autres options et choisissez Créer un compartiment.

Charger un objet de test dans votre compartiment

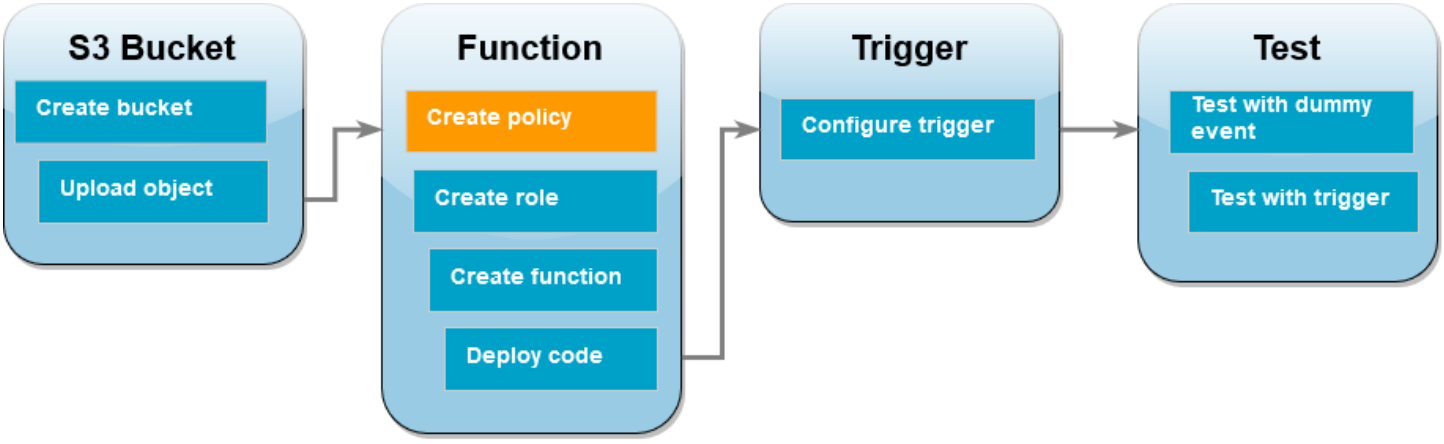


Pour charger un objet de test

1. Ouvrez la page Compartiments de la console Amazon S3 et choisissez le compartiment que vous avez créé à l'étape précédente.
2. Choisissez Charger.
3. Choisissez Ajouter des fichiers et sélectionnez l'objet que vous souhaitez charger. Vous pouvez sélectionner n'importe quel fichier (par exemple, `HappyFace.jpg`).
4. Choisissez Ouvrir, puis Charger.

Plus loin dans le tutoriel, vous testerez votre fonction Lambda à l'aide de cet objet.

Création d'une stratégie d'autorisations



Créez une politique d'autorisation qui permet à Lambda d'obtenir des objets depuis un compartiment Amazon S3 et d'écrire dans Amazon CloudWatch Logs.

Pour créer la politique

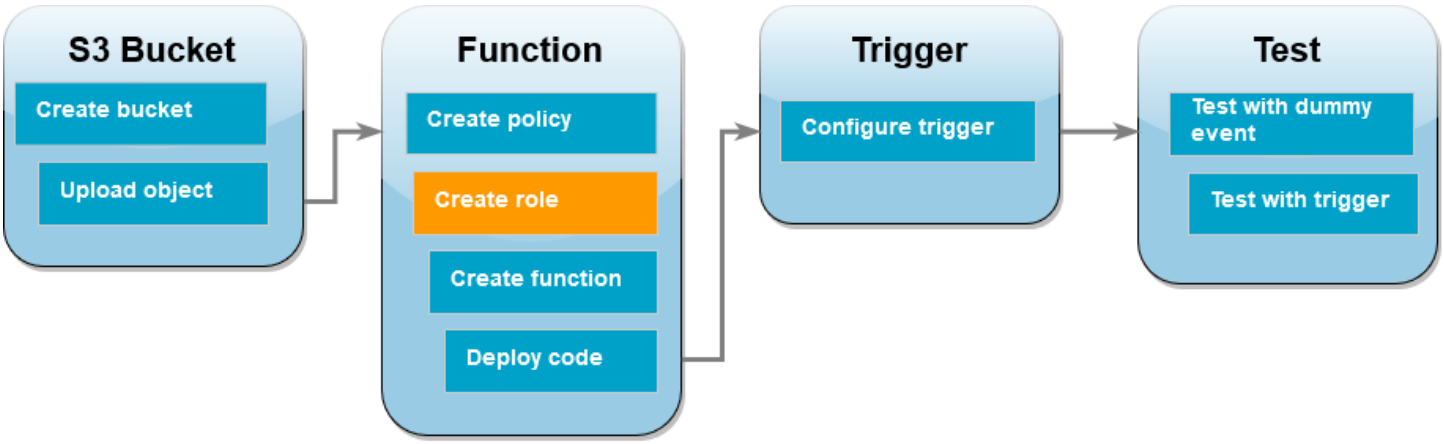
1. Ouvrez la page stratégies de la console IAM.
2. Choisissez Créer une stratégie.
3. Choisissez l'onglet JSON, puis collez la stratégie personnalisée suivante dans l'éditeur JSON.

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        }
    ]
}
```

4. Choisissez Suivant : Balises.
5. Choisissez Suivant : Vérification.
6. Sous Examiner une stratégie, pour le Nom de la stratégie, saisissez **`s3-trigger-tutorial`**.
7. Choisissez Créer une stratégie.

Créer un rôle d’exécution

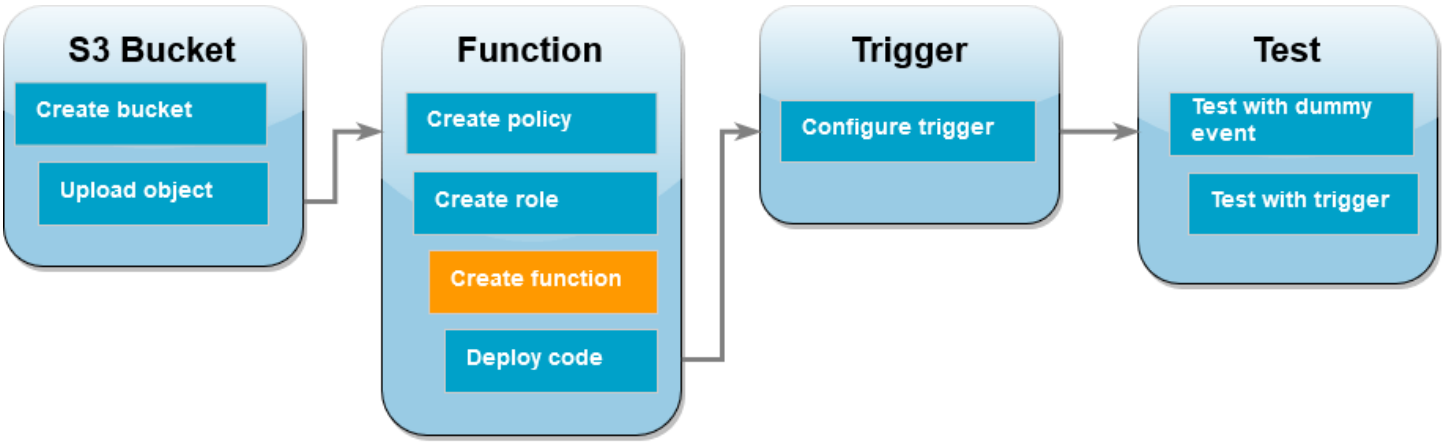


Un rôle d'exécution est un rôle Gestion des identités et des accès AWS (IAM) qui accorde à une fonction Lambda l'autorisation d' Services AWS accès et de ressources. Dans cette étape, vous créez un rôle d’exécution à l’aide de la politique d’autorisations que vous avez créée à l’étape précédente.

Pour créer un rôle d’exécution et attacher votre politique d’autorisations personnalisée

1. Ouvrez la page Rôles de la console IAM.
2. Sélectionnez Créer un rôle.
3. Pour le type d’entité de confiance, choisissez Service AWS , puis pour le cas d’utilisation, choisissez Lambda.
4. Choisissez Suivant.
5. Dans la zone de recherche de stratégie, entrez **`s3-trigger-tutorial`**.
6. Dans les résultats de la recherche, sélectionnez la stratégie que vous avez créée (`s3-trigger-tutorial`), puis choisissez Suivant.
7. Sous Role details (Détails du rôle), pour Role name (Nom du rôle), saisissez **`lambda-s3-trigger-role`**, puis sélectionnez Create role (Créer un rôle).

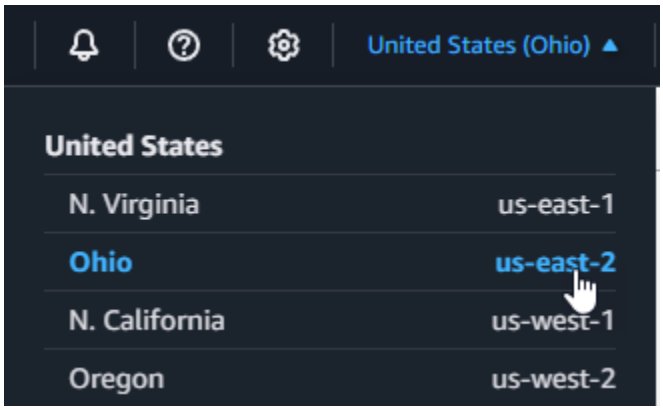
Créer la fonction Lambda



Créez une fonction Lambda dans la console à l'aide de l'environnement d'exécution Python 3.14.

Pour créer la fonction Lambda

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Assurez-vous de travailler dans le même environnement que celui dans Région AWS lequel vous avez créé votre compartiment Amazon S3. Vous pouvez modifier votre région à l'aide de la liste déroulante en haut de l'écran.



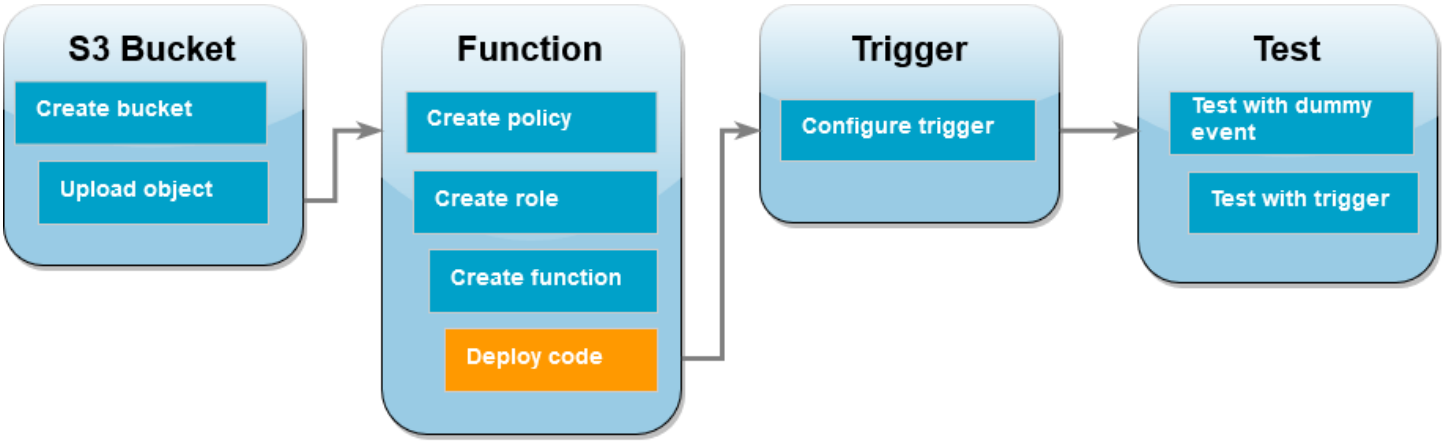
3. Choisissez Créer une fonction.
4. Choisissez Créer à partir de zéro.
5. Sous Basic information (Informations de base), procédez comme suit :
   a. Sous Nom de la fonction, saisissez `s3-trigger-tutorial`.
   b. Pour Runtime, choisissez Python 3.14.
   c. Pour Architecture, choisissez x86_64.
6. Dans l'onglet Modifier le rôle d'exécution par défaut, procédez comme suit :

a. Ouvrez l'onglet, puis choisissez Utiliser un rôle existant.

b. Sélectionnez le `lambda-s3-trigger-role` que vous avez créé précédemment.

7. Choisissez Créer une fonction.

Déployer le code de la fonction



Ce didacticiel utilise le moteur d'exécution Python 3.14, mais nous avons également fourni des exemples de fichiers de code pour d'autres environnements d'exécution. Vous pouvez sélectionner l'onglet dans la zone suivante pour voir le code d'exécution qui vous intéresse.

La fonction Lambda récupère le nom de la clé de l'objet chargé et le nom du compartiment à partir du paramètre `event` qu'elle reçoit d'Amazon S3. La fonction utilise ensuite la méthode get_object du AWS SDK pour Python (Boto3) pour récupérer les métadonnées de l'objet, y compris le type de contenu (type MIME) de l'objet chargé.

Pour déployer le code de la fonction

1. Choisissez l'onglet Python dans la zone suivante et copiez le code.

   .NET

   SDK pour .NET

> **Note**
>
> Il y en a plus sur GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement S3 avec Lambda en utilisant .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.S3;
using System;
using Amazon.Lambda.S3Events;
using System.Web;

// Assembly attribute to enable the Lambda function's JSON input to be
 converted into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJson

namespace S3Integration
{
    public class Function
    {
        private static AmazonS3Client _s3Client;
        public Function() : this(null)
        {
        }

        internal Function(AmazonS3Client s3Client)
        {
            _s3Client = s3Client ?? new AmazonS3Client();
        }

        public async Task<string> Handler(S3Event evt, ILambdaContext
 context)
        {
            try
            {
                if (evt.Records.Count <= 0)
                {
                    context.Logger.LogLine("Empty S3 Event received");
                    return string.Empty;
                }

                var bucket = evt.Records[0].S3.Bucket.Name;
```

```
                var key =
 HttpUtility.UrlDecode(evt.Records[0].S3.Object.Key);

                context.Logger.LogLine($"Request is for {bucket} and {key}");

                var objectResult = await _s3Client.GetObjectAsync(bucket,
 key);

                context.Logger.LogLine($"Returning {objectResult.Key}");

                return objectResult.Key;
            }
            catch (Exception e)
            {
                context.Logger.LogLine($"Error processing request -
 {e.Message}");

                return string.Empty;
            }
        }
    }
}
```

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus sur GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement S3 avec Lambda en utilisant Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
```

```
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/s3"
)

func handler(ctx context.Context, s3Event events.S3Event) error {
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Printf("failed to load default config: %s", err)
  return err
 }
 s3Client := s3.NewFromConfig(sdkConfig)

 for _, record := range s3Event.Records {
  bucket := record.S3.Bucket.Name
  key := record.S3.Object.URLDecodedKey
  headOutput, err := s3Client.HeadObject(ctx, &s3.HeadObjectInput{
   Bucket: &bucket,
   Key:    &key,
  })
  if err != nil {
   log.Printf("error getting head of object %s/%s: %s", bucket, key, err)
   return err
  }
  log.Printf("successfully retrieved %s/%s of type %s", bucket, key,
 *headOutput.ContentType)
 }

 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus sur GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement S3 avec Lambda en utilisant Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import software.amazon.awssdk.services.s3.model.HeadObjectRequest;
import software.amazon.awssdk.services.s3.model.HeadObjectResponse;
import software.amazon.awssdk.services.s3.S3Client;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;
import
 com.amazonaws.services.lambda.runtime.events.models.s3.S3EventNotification.S3EventNo

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Handler implements RequestHandler<S3Event, String> {
    private static final Logger logger =
 LoggerFactory.getLogger(Handler.class);
    @Override
    public String handleRequest(S3Event s3event, Context context) {
        try {
          S3EventNotificationRecord record = s3event.getRecords().get(0);
          String srcBucket = record.getS3().getBucket().getName();
          String srcKey = record.getS3().getObject().getUrlDecodedKey();

          S3Client s3Client = S3Client.builder().build();
```

```
            HeadObjectResponse headObject = getHeadObject(s3Client, srcBucket,
 srcKey);

            logger.info("Successfully retrieved " + srcBucket + "/" + srcKey +
 " of type " + headObject.contentType());

            return "Ok";
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    private HeadObjectResponse getHeadObject(S3Client s3Client, String
 bucket, String key) {
        HeadObjectRequest headObjectRequest = HeadObjectRequest.builder()
                .bucket(bucket)
                .key(key)
                .build();
        return s3Client.headObject(headObjectRequest);
    }
}
```

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus sur GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement S3 avec Lambda en utilisant. JavaScript

```
import { S3Client, HeadObjectCommand } from "@aws-sdk/client-s3";

const client = new S3Client();

export const handler = async (event, context) => {

    // Get the object from the event and show its content type
```

```
    const bucket = event.Records[0].s3.bucket.name;
    const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\
+/g, ' '));

    try {
        const { ContentType } = await client.send(new HeadObjectCommand({
            Bucket: bucket,
            Key: key,
        }));

        console.log('CONTENT TYPE:', ContentType);
        return ContentType;

    } catch (err) {
        console.log(err);
        const message = `Error getting object ${key} from bucket ${bucket}.
 Make sure they exist and your bucket is in the same region as this
 function.`;
        console.log(message);
        throw new Error(message);
    }
};
```

Consommation d'un événement S3 avec Lambda en utilisant. TypeScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { S3Event } from 'aws-lambda';
import { S3Client, HeadObjectCommand } from '@aws-sdk/client-s3';

const s3 = new S3Client({ region: process.env.AWS_REGION });

export const handler = async (event: S3Event): Promise<string | undefined> =>
 {
  // Get the object from the event and show its content type
  const bucket = event.Records[0].s3.bucket.name;
  const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/
g, ' '));
  const params = {
    Bucket: bucket,
    Key: key,
  };
```

```
  try {
    const { ContentType } = await s3.send(new HeadObjectCommand(params));
    console.log('CONTENT TYPE:', ContentType);
    return ContentType;
  } catch (err) {
    console.log(err);
    const message = `Error getting object ${key} from bucket ${bucket}. Make
 sure they exist and your bucket is in the same region as this function.`;
    console.log(message);
    throw new Error(message);
  }
};
```

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus sur GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement S3 avec Lambda à l'aide de PHP.

```
<?php

use Bref\Context\Context;
use Bref\Event\S3\S3Event;
use Bref\Event\S3\S3Handler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';


class Handler extends S3Handler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
```

```
    }

    public function handleS3(S3Event $event, Context $context) : void
    {
        $this->logger->info("Processing S3 records");

        // Get the object from the event and show its content type
        $records = $event->getRecords();

        foreach ($records as $record)
        {
            $bucket = $record->getBucket()->getName();
            $key = urldecode($record->getObject()->getKey());

            try {
                $fileSize = urldecode($record->getObject()->getSize());
                echo "File Size: " . $fileSize . "\n";
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                echo $e->getMessage() . "\n";
                echo 'Error getting object ' . $key . ' from bucket ' .
 $bucket . '. Make sure they exist and your bucket is in the same region as
 this function.' . "\n";
                throw $e;
            }
        }
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus sur GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement S3 avec Lambda en utilisant Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import json
import urllib.parse
import boto3

print('Loading function')

s3 = boto3.client('s3')


def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))

    # Get the object from the event and show its content type
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']
['key'], encoding='utf-8')
    try:
        response = s3.get_object(Bucket=bucket, Key=key)
        print("CONTENT TYPE: " + response['ContentType'])
        return response['ContentType']
    except Exception as e:
        print(e)
        print('Error getting object {} from bucket {}. Make sure they
 exist and your bucket is in the same region as this function.'.format(key,
 bucket))
        raise e
```

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus sur GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement S3 avec Lambda à l'aide de Ruby.

```
require 'json'
require 'uri'
require 'aws-sdk'

puts 'Loading function'

def lambda_handler(event:, context:)
  s3 = Aws::S3::Client.new(region: 'region') # Your AWS region
  # puts "Received event: #{JSON.dump(event)}"

  # Get the object from the event and show its content type
  bucket = event['Records'][0]['s3']['bucket']['name']
  key = URI.decode_www_form_component(event['Records'][0]['s3']['object']
['key'], Encoding::UTF_8)
  begin
    response = s3.get_object(bucket: bucket, key: key)
    puts "CONTENT TYPE: #{response.content_type}"
    return response.content_type
  rescue StandardError => e
    puts e.message
    puts "Error getting object #{key} from bucket #{bucket}. Make sure they
 exist and your bucket is in the same region as this function."
    raise e
  end
end
```

Rust

SDK pour Rust

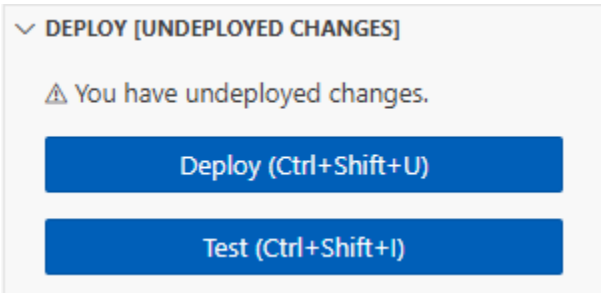
> **Note**
>
> Il y en a plus sur GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement S3 avec Lambda en utilisant Rust.

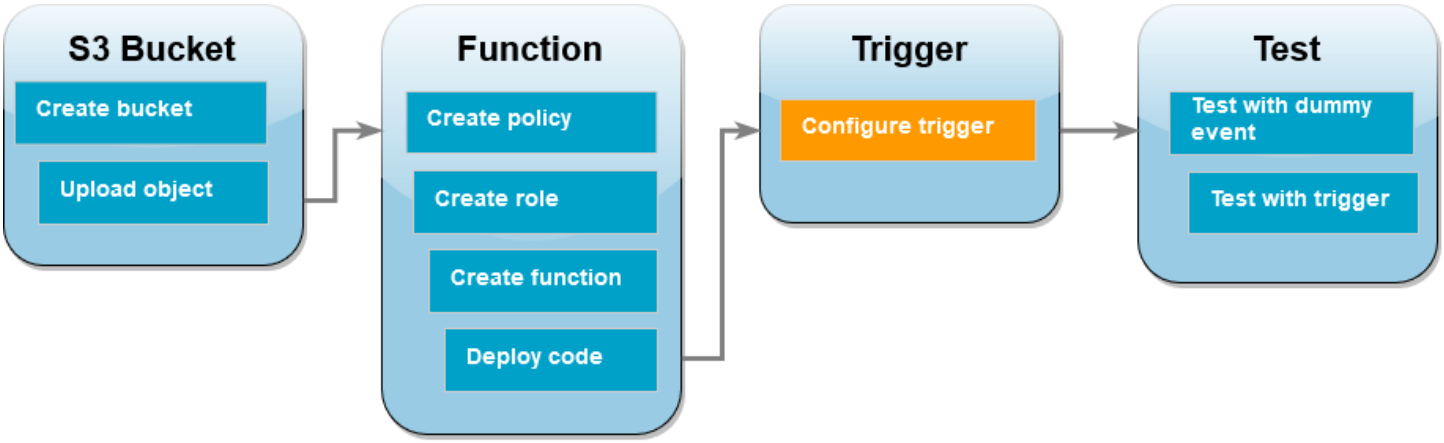
```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::s3::S3Event;
use aws_sdk_s3::{Client};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};


/// Main function
#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    // Initialize the AWS SDK for Rust
    let config = aws_config::load_from_env().await;
    let s3_client = Client::new(&config);

    let res = run(service_fn(|request: LambdaEvent<S3Event>| {
        function_handler(&s3_client, request)
    })).await;

    res
}

async fn function_handler(
    s3_client: &Client,
```

```
    evt: LambdaEvent<S3Event>
) -> Result<(), Error> {
    tracing::info!(records = ?evt.payload.records.len(), "Received request
 from SQS");

    if evt.payload.records.len() == 0 {
        tracing::info!("Empty S3 event received");
    }

    let bucket =
 evt.payload.records[0].s3.bucket.name.as_ref().expect("Bucket name to
 exist");
    let key = evt.payload.records[0].s3.object.key.as_ref().expect("Object
 key to exist");

    tracing::info!("Request is for {} and object {}", bucket, key);

    let s3_get_object_result = s3_client
        .get_object()
        .bucket(bucket)
        .key(key)
        .send()
        .await;

    match s3_get_object_result {
        Ok(_) => tracing::info!("S3 Get Object success, the s3GetObjectResult
 contains a 'body' property of type ByteStream"),
        Err(_) => tracing::info!("Failure with S3 Get Object request")
    }

    Ok(())
}
```

2. Dans le volet Code source de la console Lambda, collez le code dans l'éditeur de code, en remplaçant le code créé par Lambda.
3. Dans la section DÉPLOYER, choisissez Déployer pour mettre à jour le code de votre fonction :

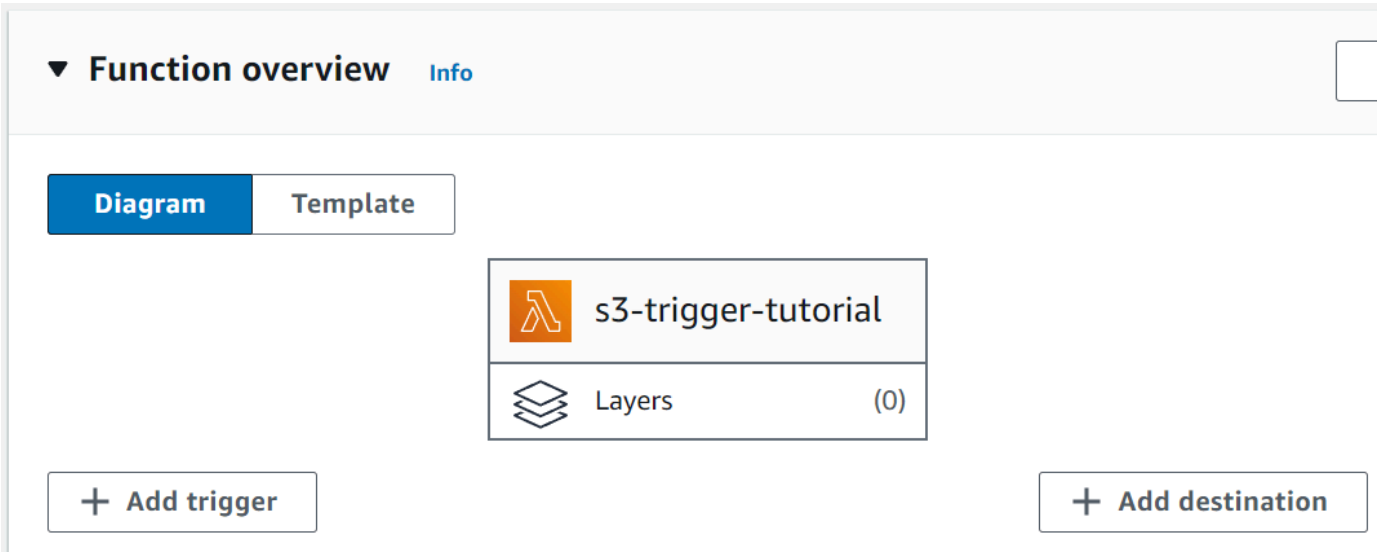


Création d'un déclencheur Amazon S3



Pour créer le déclencheur Amazon S3

1. Dans le volet de Présentation de la fonction, choisissez Ajouter un déclencheur.

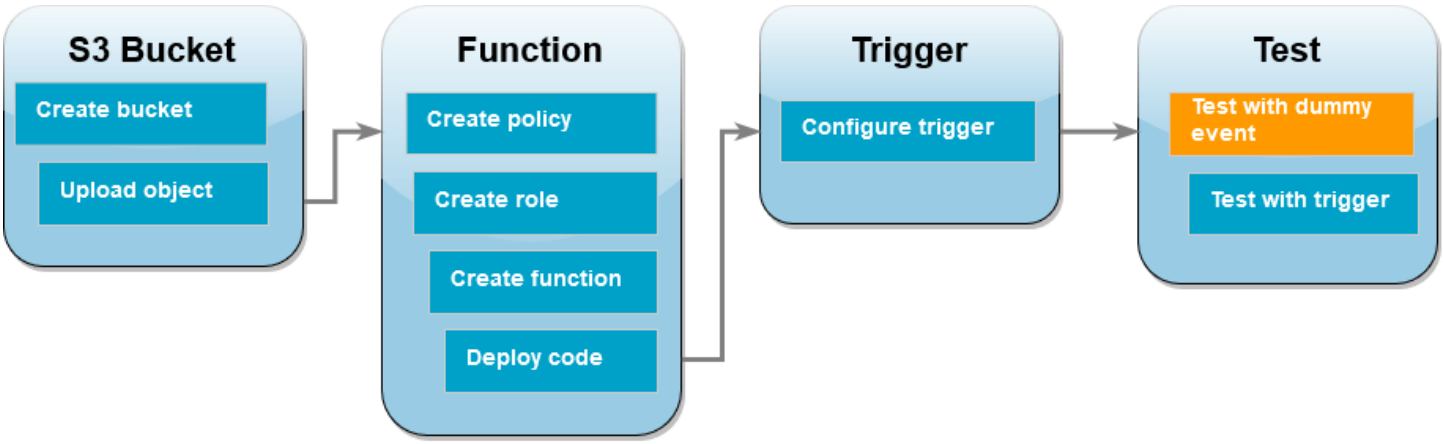


2. Sélectionnez S3.

3. Sous Compartiment, sélectionnez le compartiment que vous avez créé précédemment dans le didacticiel.
4. Sous Types d'événements, assurez-vous que Tous les événements de création d'objet est sélectionné.
5. Sous Invocation récursive, cochez la case pour confirmer qu'il n'est pas recommandé d'utiliser le même compartiment Amazon S3 pour les entrées et les sorties.
6. Choisissez Ajouter.

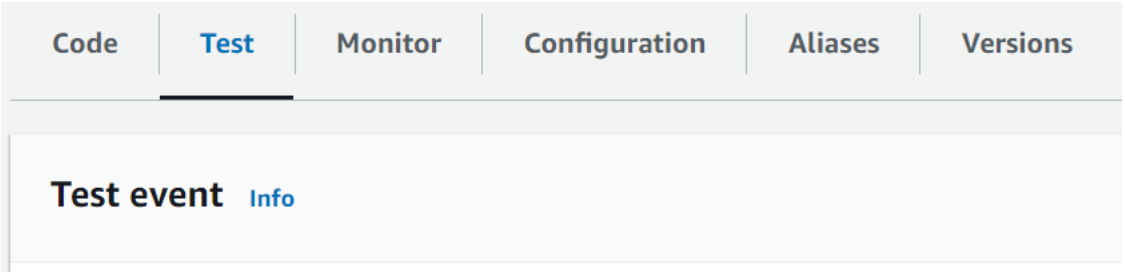
> **Note**
>
> Lorsque vous créez un déclencheur Amazon S3 pour une fonction Lambda à l'aide de la console Lambda, Amazon S3 configure une notification d'événement sur le compartiment que vous spécifiez. Avant de configurer cette notification d'événement, Amazon S3 effectue une série de vérifications pour confirmer que la destination de l'événement existe et dispose des politiques IAM requises. Amazon S3 effectue également ces tests sur toutes les autres notifications d'événements configurées pour ce compartiment.
> En raison de cette vérification, si le compartiment a déjà configuré des destinations d'événements pour des ressources qui n'existent plus ou pour des ressources qui ne disposent pas des politiques d'autorisations requises, Amazon S3 ne sera pas en mesure de créer la nouvelle notification d'événement. Vous verrez le message d'erreur suivant indiquant que votre déclencheur n'a pas pu être créé :
>
> ```
> An error occurred when creating the trigger: Unable to validate the following
>  destination configurations.
> ```
>
> Vous pouvez voir cette erreur si vous avez précédemment configuré un déclencheur pour une autre fonction Lambda utilisant le même compartiment, et si vous avez depuis supprimé la fonction ou modifié ses politiques d'autorisations.

Test de votre fonction Lambda à l'aide d'un événement fictif



Pour tester la fonction Lambda à l'aide d'un événement fictif

1. Dans la page de votre fonction de la console Lambda, choisissez l'onglet Tester.



2. Dans Event name (Nom de l'événement), saisissez `MyTestEvent`.
3. Dans le JSON d'événement, collez l'événement de test suivant. Veillez à remplacer les valeurs suivantes :

   - Remplacez `us-east-1` par la région dans laquelle vous avez créé votre compartiment Amazon S3.
   - Remplacez les deux instances de `amzn-s3-demo-bucket` par le nom de votre propre compartiment Amazon S3.
   - Remplacez `test%2FKey` par le nom de l'objet de test que vous avez chargé précédemment dans votre compartiment (par exemple, `HappyFace.jpg`).

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
```

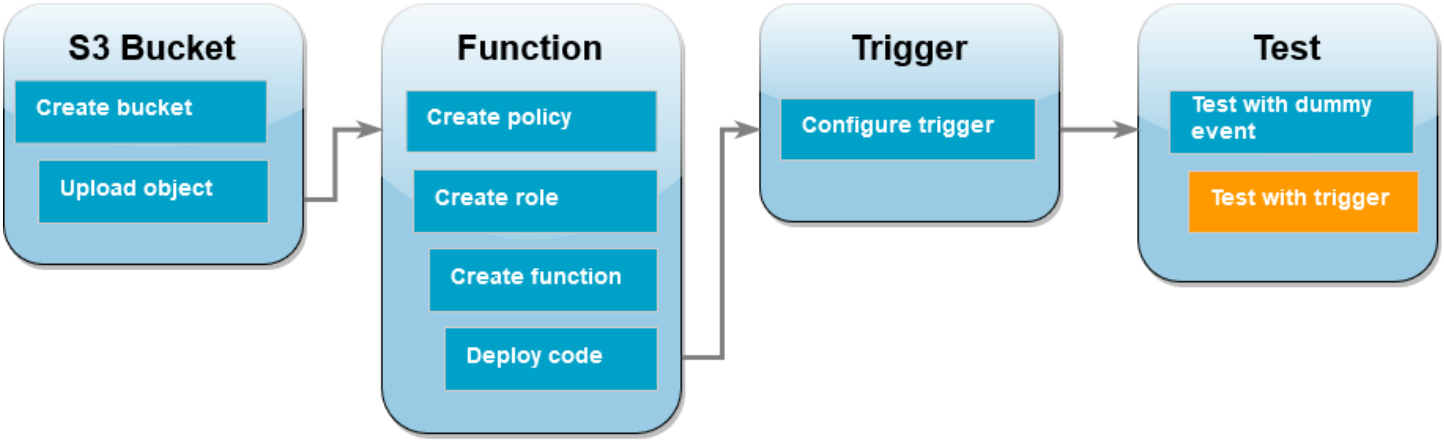
```
      "awsRegion": "us-east-1",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "EXAMPLE123456789",
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/
mnopqrstuvwxyzABCDEFGH"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "amzn-s3-demo-bucket",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          },
          "arn": "arn:aws:s3:::amzn-s3-demo-bucket"
        },
        "object": {
          "key": "test%2Fkey",
          "size": 1024,
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901"
        }
      }
    }
  ]
}
```

4. Choisissez Enregistrer.
5. Sélectionnez Tester).
6. Si votre fonction s'exécute correctement, vous obtiendrez un résultat similaire à celui qui suit dans l'onglet Résultats de l'exécution.

```
Response
"image/jpeg"
```

```
Function Logs
START RequestId: 12b3cae7-5f4e-415e-93e6-416b8f8b66e6 Version: $LATEST
2021-02-18T21:40:59.280Z    12b3cae7-5f4e-415e-93e6-416b8f8b66e6    INFO    INPUT
 BUCKET AND KEY:  { Bucket: 'amzn-s3-demo-bucket', Key: 'HappyFace.jpg' }
2021-02-18T21:41:00.215Z    12b3cae7-5f4e-415e-93e6-416b8f8b66e6    INFO    CONTENT
 TYPE: image/jpeg
END RequestId: 12b3cae7-5f4e-415e-93e6-416b8f8b66e6
REPORT RequestId: 12b3cae7-5f4e-415e-93e6-416b8f8b66e6    Duration: 976.25 ms
 Billed Duration: 977 ms    Memory Size: 128 MB    Max Memory Used: 90 MB    Init
 Duration: 430.47 ms

Request ID
12b3cae7-5f4e-415e-93e6-416b8f8b66e6
```

Test de la fonction Lambda avec le déclencheur Amazon S3



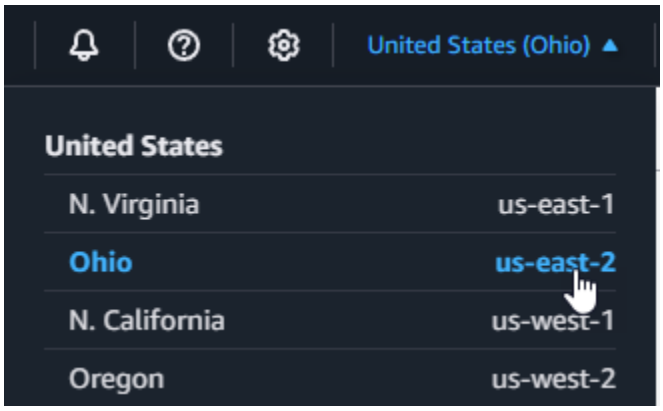
Pour tester votre fonction avec le déclencheur configuré, chargez un objet dans votre compartiment Amazon S3 à l'aide de la console. Pour vérifier que votre fonction Lambda s'est exécutée comme prévu, utilisez CloudWatch Logs pour afficher le résultat de votre fonction.

Pour charger un objet dans votre compartiment Amazon S3

1. Ouvrez la page Compartiments de la console Amazon S3 et choisissez le compartiment que vous avez créé précédemment.
2. Choisissez Charger.
3. Choisissez Ajouter des fichiers et utilisez le sélecteur de fichiers pour choisir l'objet que vous souhaitez charger. Cet objet peut être n'importe quel fichier que vous choisissez.
4. Choisissez Ouvrir, puis Charger.

Pour vérifier l'invocation de la fonction à l'aide CloudWatch de Logs

1. Ouvrez la console CloudWatch.
2. Assurez-vous de travailler de la même manière que celle dans laquelle Région AWS vous avez créé votre fonction Lambda. Vous pouvez modifier votre région à l'aide de la liste déroulante en haut de l'écran.



3. Choisissez Journaux, puis Groupes de journaux.
4. Choisissez le groupe de journaux de votre fonction (`/aws/lambda/s3-trigger-tutorial`).
5. Sous Flux de journaux, sélectionnez le flux de journaux le plus récent.
6. Si votre fonction a été invoquée correctement en réponse à votre déclencheur Amazon S3, vous obtiendrez une sortie similaire à celle qui suit. Le `CONTENT TYPE` que vous voyez dépend du type de fichier que vous avez chargé dans votre compartiment.

```
2022-05-09T23:17:28.702Z 0cae7f5a-b0af-4c73-8563-a3430333cc10 INFO CONTENT
 TYPE: image/jpeg
```

Nettoyage de vos ressources

Vous pouvez maintenant supprimer les ressources que vous avez créées pour ce didacticiel, sauf si vous souhaitez les conserver. En supprimant AWS les ressources que vous n'utilisez plus, vous évitez des frais inutiles pour votre Compte AWS.

Pour supprimer la fonction Lambda

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction que vous avez créée.
3. Sélectionnez Actions, Supprimer.

4. Saisissez **`confirm`** dans la zone de saisie de texte et choisissez Delete (Supprimer).

Pour supprimer le rôle d'exécution

1. Ouvrez la page Roles (Rôles) de la console IAM.
2. Sélectionnez le rôle d'exécution que vous avez créé.
3. Sélectionnez Delete (Supprimer).
4. Saisissez le nom du rôle dans le champ de saisie de texte et choisissez Delete (Supprimer).

Pour supprimer le compartiment S3

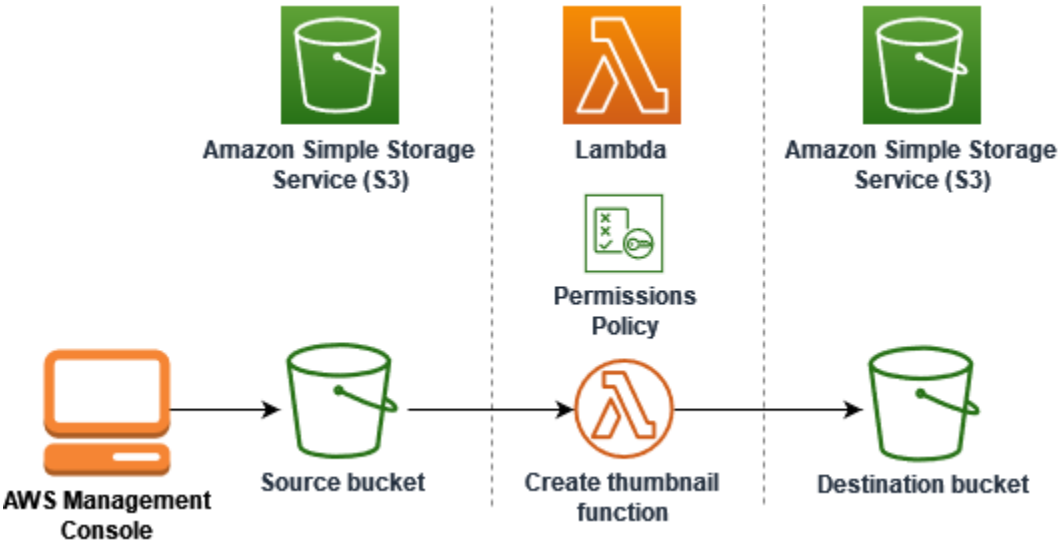
1. Ouvrez la console Amazon S3.
2. Sélectionnez le compartiment que vous avez créé.
3. Sélectionnez Delete (Supprimer).
4. Saisissez le nom du compartiment dans le champ de saisie de texte.
5. Choisissez Supprimer le compartiment.

Étapes suivantes

Dans Didacticiel : Utilisation d'un déclencheur Amazon S3 pour créer des images miniatures, le déclencheur Amazon S3 invoque une fonction qui crée une image de miniature pour chaque fichier image qui est chargé dans votre compartiment. Ce didacticiel nécessite un niveau modéré de connaissance du AWS domaine Lambda. Il montre comment créer des ressources à l'aide de AWS Command Line Interface (AWS CLI) et comment créer un package de déploiement d'archives de fichiers .zip pour la fonction et ses dépendances.

Didacticiel : Utilisation d'un déclencheur Amazon S3 pour créer des images miniatures

Dans ce tutoriel, vous allez créer et configurer une fonction Lambda qui redimensionne les images ajoutées à un compartiment Amazon Simple Storage Service (Amazon S3). Lorsque vous ajoutez un fichier image à votre compartiment, Amazon S3 invoque votre fonction Lambda. La fonction crée ensuite une version miniature de l'image et l'envoie vers un autre compartiment Amazon S3.



Pour compléter ce didacticiel, effectuez les tâches suivantes :

1. Créez des compartiments Amazon S3 source et de destination et chargez un exemple d'image.
2. Créez une fonction Lambda qui redimensionne une image et envoie une miniature vers un compartiment Amazon S3.
3. Configurez un déclencheur Lambda qui invoque votre fonction lorsque des objets sont chargés dans votre compartiment source.
4. Testez votre fonction, d'abord avec un événement fictif, puis en chargeant une image dans votre compartiment source.

En suivant ces étapes, vous apprendrez à utiliser Lambda pour effectuer une tâche de traitement de fichiers sur des objets ajoutés à un compartiment Amazon S3. Vous pouvez suivre ce didacticiel en utilisant le AWS Command Line Interface (AWS CLI) ou le AWS Management Console.

Si vous recherchez un exemple plus simple pour apprendre à configurer un déclencheur Amazon S3 pour Lambda, vous pouvez consulter le Didacticiel : utilisation d'un déclencheur Amazon S3 pour invoquer une fonction Lambda.

Rubriques

- Conditions préalables
- Création de deux compartiments Amazon S3
- Charger une image de test dans votre compartiment source
- Création d'une stratégie d'autorisations

- Créer un rôle d'exécution
- Créer le package de déploiement de la fonction
- Créer la fonction Lambda
- Configurer Amazon S3 pour invoquer la fonction
- Test de votre fonction Lambda à l'aide d'un événement fictif
- Tester votre fonction à l'aide du déclencheur Amazon S3
- Nettoyage de vos ressources

Conditions préalables

Si vous souhaitez utiliser le AWS CLI pour terminer le didacticiel, installez la dernière version du AWS Command Line Interface.

Pour le code de votre fonction Lambda, vous pouvez utiliser Python ou Node.js. Installez les outils de prise en charge linguistique et un gestionnaire de packages pour le langage que vous souhaitez utiliser.

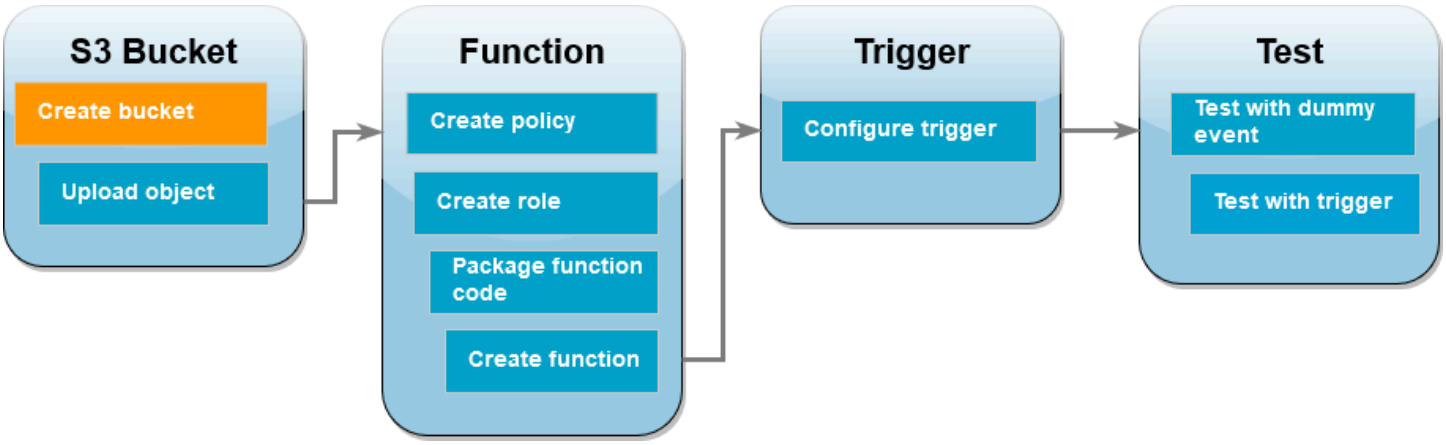
Installez le AWS Command Line Interface

Si vous ne l'avez pas encore installé AWS Command Line Interface, suivez les étapes décrites dans la section Installation ou mise à jour de la dernière version du AWS CLI pour l'installer.

Ce tutoriel nécessite un terminal de ligne de commande ou un shell pour exécuter les commandes. Sous Linux et macOS, utilisez votre gestionnaire de shell et de package préféré.

> **Note**
>
> Sous Windows, certaines commandes CLI Bash que vous utilisez couramment avec Lambda (par exemple `zip`) ne sont pas prises en charge par les terminaux intégrés du système d'exploitation. Installez le sous-système Windows pour Linux afin d'obtenir une version intégrée à Windows d'Ubuntu et Bash.

Création de deux compartiments Amazon S3

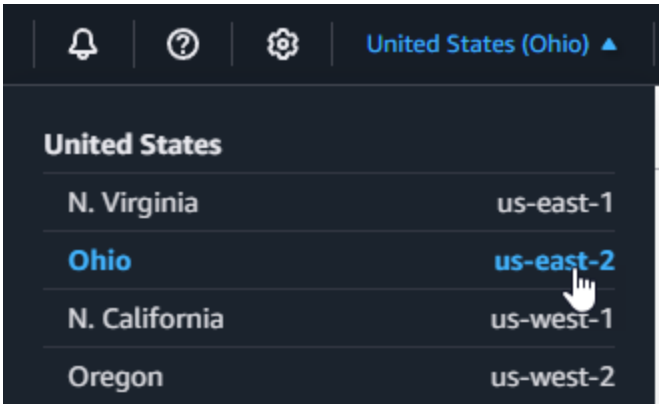


Créez d'abord deux compartiments Amazon S3. Le premier compartiment est le compartiment source dans lequel vous allez charger vos images. Le second compartiment est utilisé par Lambda pour enregistrer la miniature redimensionnée lorsque vous invoquez votre fonction.

AWS Management Console

Pour créer les compartiments Amazon S3 (console)

1. Ouvrez la console Amazon S3 et sélectionnez la page Compartiments à usage général.
2. Sélectionnez le Région AWS plus proche de votre situation géographique. Vous pouvez modifier votre région à l'aide de la liste déroulante en haut de l'écran. Plus loin dans le didacticiel, vous devez créer votre fonction Lambda dans la même région.



3. Choisissez Create bucket (Créer un compartiment).
4. Sous Configuration générale, procédez comme suit :

   a. Pour le type de compartiment, veillez à sélectionner Usage général.

b. Pour le nom du compartiment, saisissez un nom unique au monde qui respecte les règles de dénomination du compartiment Amazon S3. Les noms de compartiment peuvent contenir uniquement des lettres minuscules, des chiffres, de points (.) et des traits d'union (-).

5. Conservez les valeurs par défaut de toutes les autres options et choisissez Créer un compartiment.
6. Répétez les étapes 1 à 5 pour créer votre compartiment de destination. Pour Nom du compartiment, saisissez **`amzn-s3-demo-source-bucket-resized`**, où **`amzn-s3-demo-source-bucket`** est le nom du compartiment source que vous venez de créer.

AWS CLI

Pour créer les compartiments Amazon S3 (AWS CLI)

1. Exécutez la commande CLI suivante pour créer votre compartiment source. Le nom que vous choisissez pour votre compartiment doit être unique au monde et respecter les Règles de dénomination du compartiment Amazon S3. Les noms peuvent contenir uniquement des lettres minuscules, des chiffres, de points (.) et des traits d'union (-). Pour `region` et `LocationConstraint`, choisissez la Région AWS la plus proche de votre emplacement géographique.

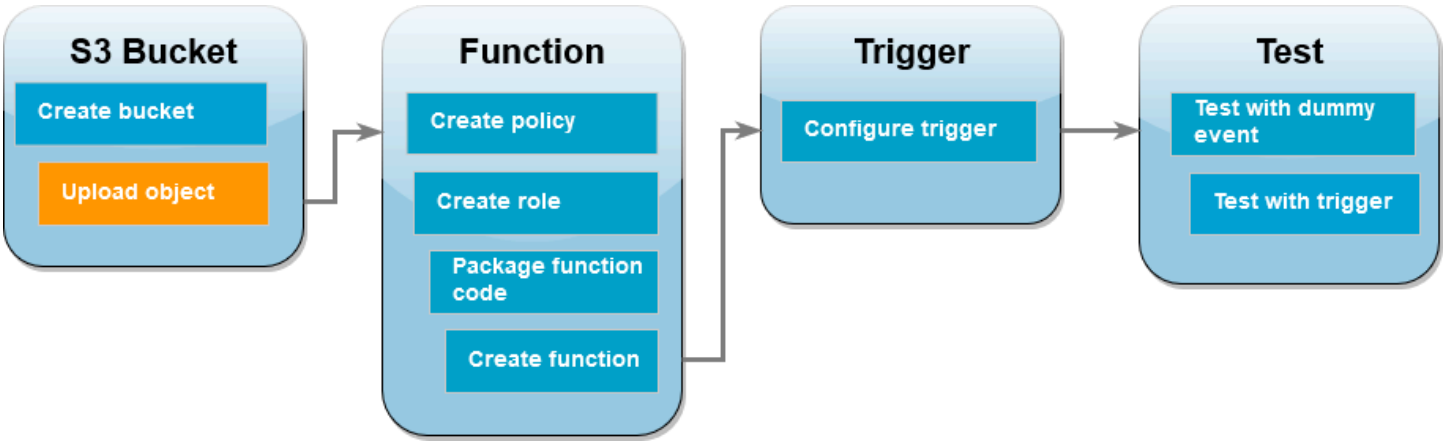
```
aws s3api create-bucket --bucket amzn-s3-demo-source-bucket --region us-east-1 \
--create-bucket-configuration LocationConstraint=us-east-1
```

Plus loin dans le didacticiel, vous devez créer votre fonction Lambda dans le même Région AWS emplacement que votre compartiment source. Notez donc la région que vous avez choisie.

2. Exécutez la commande suivante pour créer votre compartiment de destination. Pour le nom du compartiment, vous devez utiliser **`amzn-s3-demo-source-bucket-resized`**, où **`amzn-s3-demo-source-bucket`** est le nom du compartiment source que vous avez créé à l'étape 1. Pour `region` et`LocationConstraint`, choisissez le même Région AWS que celui que vous avez utilisé pour créer votre bucket source.

```
aws s3api create-bucket --bucket amzn-s3-demo-source-bucket-resized --region us-
east-1 \
--create-bucket-configuration LocationConstraint=us-east-1
```

Charger une image de test dans votre compartiment source



Plus loin dans le didacticiel, vous testerez votre fonction Lambda en l'invoquant à l'aide de la console Lambda ou de AWS CLI la console Lambda. Pour confirmer que votre fonction fonctionne correctement, votre compartiment source doit contenir une image de test. Cette image peut être n'importe quel fichier JPG ou PNG de votre choix.

AWS Management Console

Pour charger une image de test dans votre compartiment source (console)

1. Ouvrez la page Compartiments de la console Amazon S3.
2. Sélectionnez le compartiment source que vous avez créé à l'étape précédente.
3. Choisissez Charger.
4. Choisissez Ajouter des fichiers et utilisez le sélecteur de fichiers pour sélectionner l'objet que vous souhaitez charger.
5. Choisissez Ouvrir, puis Charger.

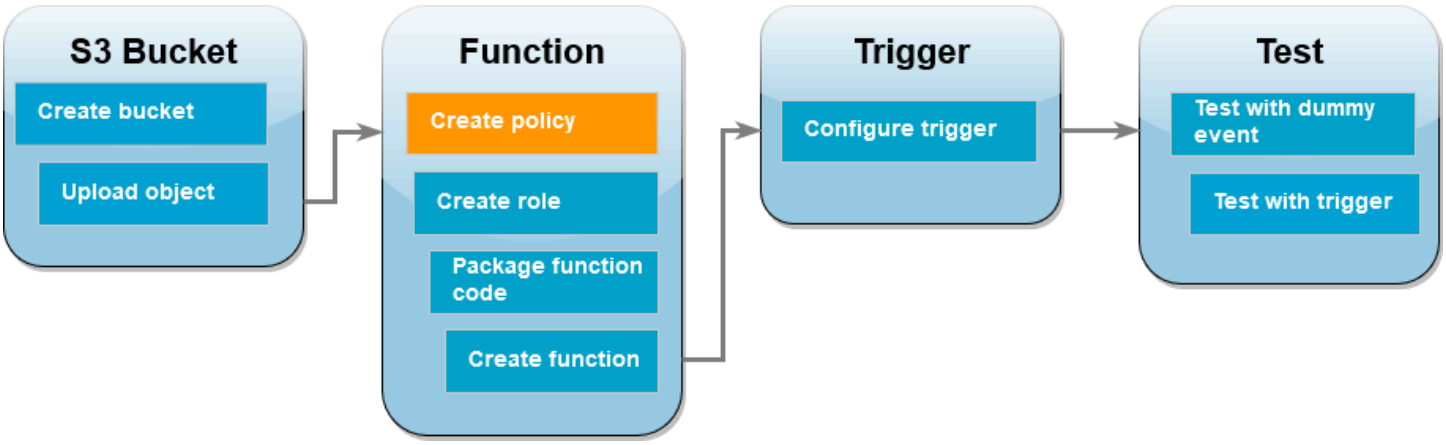
AWS CLI

Pour charger une image de test dans votre compartiment source (AWS CLI)

- À partir du répertoire contenant l'image que vous souhaitez charger, exécutez la commande CLI suivante. Remplacez le paramètre `--bucket` par le nom de votre compartiment source. Pour les paramètres `--key` et `--body`, utilisez le nom de fichier de votre image de test.

```
aws s3api put-object --bucket amzn-s3-demo-source-bucket --key HappyFace.jpg --
body ./HappyFace.jpg
```

Création d'une stratégie d'autorisations



La première étape de la création de votre fonction Lambda consiste à créer une politique d'autorisations. Cette politique donne à votre fonction les autorisations dont elle a besoin pour accéder à d'autres AWS ressources. Pour ce didacticiel, la politique donne à Lambda des autorisations de lecture et d'écriture pour les compartiments Amazon S3 et lui permet d'écrire dans Amazon Logs. CloudWatch

AWS Management Console

Pour créer une politique (console)

1. Ouvrez la page Politiques de la console Gestion des identités et des accès AWS (IAM).
2. Choisissez Create Policy (Créer une politique).
3. Choisissez l'onglet JSON, puis collez la stratégie personnalisée suivante dans l'éditeur JSON.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
```

```
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        }
    ]
}
```

4. Choisissez Suivant.
5. Sous Détails de la politique, pour le Nom de la politique, saisissez **`LambdaS3Policy`**.
6. Choisissez Create Policy (Créer une politique).

AWS CLI

Pour créer la politique (AWS CLI)

1. Enregistrez le JSON suivant dans un fichier nommé `policy.json`.

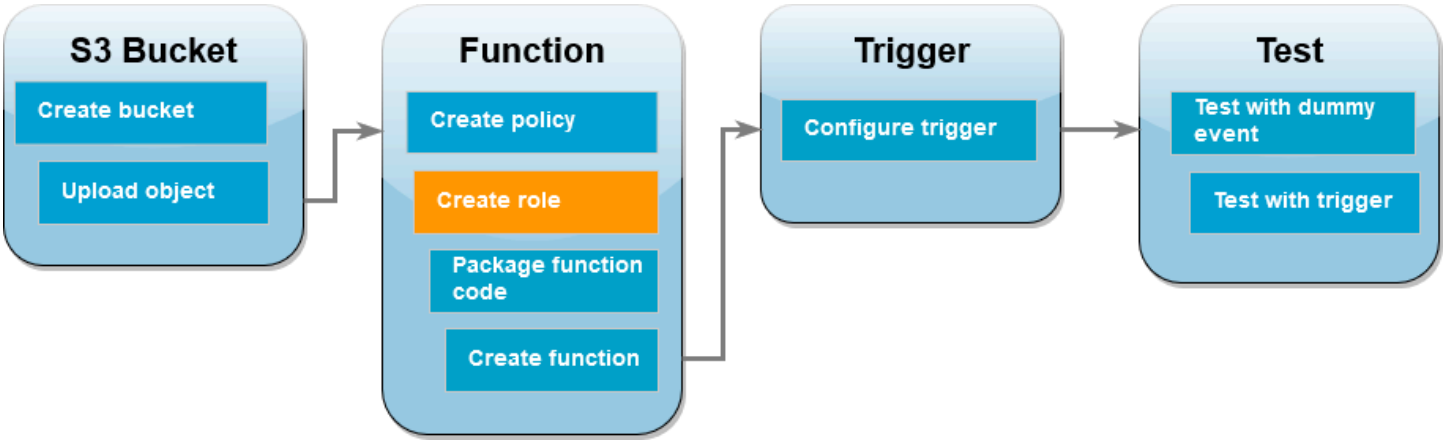
```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
```

```
            "Action": [
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        }
    ]
}
```

2. À partir du répertoire dans lequel vous avez enregistré le document de politique JSON, exécutez la commande CLI suivante.

```
aws iam create-policy --policy-name LambdaS3Policy --policy-document file://
policy.json
```

Créer un rôle d’exécution



Un rôle d'exécution est un rôle IAM qui accorde à une fonction Lambda l'autorisation d' Services AWS accès et de ressources. Pour donner à votre fonction un accès en lecture et en écriture à un compartiment Amazon S3, vous attachez la politique d’autorisation que vous avez créée à l’étape précédente.

AWS Management Console

Pour créer un rôle d'exécution et attacher votre politique d'autorisations (console)

1. Accédez à la Page Rôles de la console (IAM).
2. Choisissez Créer un rôle.
3. Pour Type d'entité fiable, sélectionnez Service AWS, et pour Cas d'utilisation, sélectionnez Lambda.
4. Choisissez Suivant.
5. Ajoutez la politique d'autorisations que vous avez créée à l'étape précédente en procédant comme suit :
   a. Dans la zone de recherche de stratégie, entrez **`LambdaS3Policy`**.
   b. Dans les résultats de la recherche, cochez la case pour `LambdaS3Policy`.
   c. Choisissez Suivant.
6. Sous Détails du rôle, pour le Nom du rôle entrez **`LambdaS3Role`**.
7. Choisissez Créer un rôle.

AWS CLI

Pour créer un rôle d'exécution et attacher votre politique d'autorisations (AWS CLI)

1. Enregistrez le JSON suivant dans un fichier nommé `trust-policy.json`. Cette politique de confiance permet à Lambda d'utiliser les autorisations du rôle en autorisant le principal `lambda.amazonaws.com` du service à appeler l'action AWS Security Token Service (AWS STS)`AssumeRole`.

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
```

```
  ]
}
```

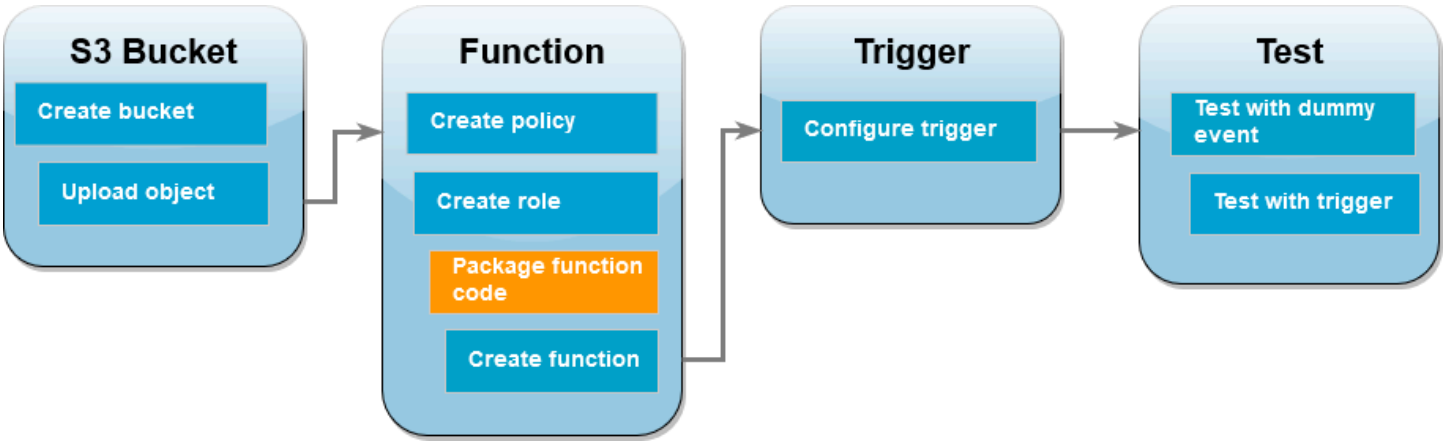
2. À partir du répertoire dans lequel vous avez enregistré le document de politique d'approbation JSON, exécutez la commande CLI suivante pour créer le rôle d'exécution.

```
aws iam create-role --role-name LambdaS3Role --assume-role-policy-document
 file://trust-policy.json
```

3. Pour attacher la politique d'autorisations que vous avez créée à l'étape précédente, exécutez la commande CLI suivante. Remplacez le Compte AWS numéro indiqué dans l'ARN de la politique par votre propre numéro de compte.

```
aws iam attach-role-policy --role-name LambdaS3Role --policy-arn
 arn:aws:iam::123456789012:policy/LambdaS3Policy
```

Créer le package de déploiement de la fonction



Pour créer votre fonction, vous créez un package de déploiement contenant le code de votre fonction et ses dépendances. Pour cette fonction `CreateThumbnail`, votre code de fonction utilise une bibliothèque distincte pour le redimensionnement de l'image. Suivez les instructions correspondant à la langue de votre choix pour créer un package de déploiement contenant la bibliothèque requise.

Node.js

Pour créer le package de déploiement (Node.js)

1. Créez un répertoire nommé `lambda-s3` pour votre code de fonction et vos dépendances et accédez-y.

```
mkdir lambda-s3
cd lambda-s3
```

2. Créez un nouveau projet Node.js avec `npm`. Appuyez sur `Enter` pour accepter les options par défaut fournies dans l'expérience interactive.

```
npm init
```

3. Enregistrez le code de fonction suivant dans un fichier nommé `index.mjs`. Assurez-vous de remplacer `us-east-1` par l' Région AWS dans lequel vous avez créé vos propres compartiments source et de destination.

```
// dependencies
import { S3Client, GetObjectCommand, PutObjectCommand } from '@aws-sdk/client-
s3';

import { Readable } from 'stream';

import sharp from 'sharp';
import util from 'util';


// create S3 client
const s3 = new S3Client({region: 'us-east-1'});

// define the handler function
export const handler = async (event, context) => {

// Read options from the event parameter and get the source bucket
console.log("Reading options from event:\n", util.inspect(event, {depth: 5}));
  const srcBucket = event.Records[0].s3.bucket.name;

// Object key may have spaces or unicode non-ASCII characters
const srcKey    = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/
g, " "));
const dstBucket = srcBucket + "-resized";
const dstKey    = "resized-" + srcKey;

// Infer the image type from the file suffix
const typeMatch = srcKey.match(/\.([^.]*)$/);
if (!typeMatch) {
  console.log("Could not determine the image type.");
```

```
  return;
}

// Check that the image type is supported
const imageType = typeMatch[1].toLowerCase();
if (imageType != "jpg" && imageType != "png") {
  console.log(`Unsupported image type: ${imageType}`);
  return;
}

// Get the image from the source bucket. GetObjectCommand returns a stream.
try {
  const params = {
    Bucket: srcBucket,
    Key: srcKey
  };
  var response = await s3.send(new GetObjectCommand(params));
  var stream = response.Body;

// Convert stream to buffer to pass to sharp resize function.
  if (stream instanceof Readable) {
    var content_buffer = Buffer.concat(await stream.toArray());

  } else {
    throw new Error('Unknown object stream type');
  }


} catch (error) {
  console.log(error);
  return;
}


// set thumbnail width. Resize will set the height automatically to maintain
 aspect ratio.
const width  = 200;

// Use the sharp module to resize the image and save in a buffer.
try {
  var output_buffer = await sharp(content_buffer).resize(width).toBuffer();

} catch (error) {
  console.log(error);
```

```
  return;
}

// Upload the thumbnail image to the destination bucket
try {
  const destparams = {
    Bucket: dstBucket,
    Key: dstKey,
    Body: output_buffer,
    ContentType: "image"
  };

  const putResult = await s3.send(new PutObjectCommand(destparams));

  } catch (error) {
    console.log(error);
    return;
  }

  console.log('Successfully resized ' + srcBucket + '/' + srcKey +
    ' and uploaded to ' + dstBucket + '/' + dstKey);
  };
```

4. Dans votre répertoire `lambda-s3`, installez la bibliothèque sharp en utilisant npm. Notez que la dernière version de sharp (0.33) n'est pas compatible avec Lambda. Installez la version 0.32.6 pour terminer ce didacticiel.

```
npm install sharp@0.32.6
```

La commande npm `install` crée un répertoire `node_modules` pour vos modules. Après cette étape, votre structure de répertoire doit se présenter comme suit.

```
lambda-s3
|- index.mjs
|- node_modules
|  |- base64js
|  |- bl
|  |- buffer
...
|- package-lock.json
|- package.json
```

5. Créez un package de déploiement .zip contenant le code de votre fonction et ses dépendances. Sous macOS ou Linux, exécutez les commandes suivantes.

```
zip -r function.zip .
```

Sous Windows, utilisez l'utilitaire zip de votre choix pour créer un fichier .zip. Assurez-vous que vos fichiers `index.mjs`, `package.json`, et `package-lock.json` ainsi que votre répertoire `node_modules` se trouvent tous à la racine de votre fichier .zip.

Python

Pour créer le package de déploiement (Python)

1. Enregistrez l'exemple de code en tant que fichier nommé `lambda_function.py`.

```
import boto3
import os
import sys
import uuid
from urllib.parse import unquote_plus
from PIL import Image
import PIL.Image

s3_client = boto3.client('s3')

def resize_image(image_path, resized_path):
  with Image.open(image_path) as image:
    image.thumbnail(tuple(x / 2 for x in image.size))
    image.save(resized_path)

def lambda_handler(event, context):
  for record in event['Records']:
    bucket = record['s3']['bucket']['name']
    key = unquote_plus(record['s3']['object']['key'])
    tmpkey = key.replace('/', '')
    download_path = '/tmp/{}{}'.format(uuid.uuid4(), tmpkey)
    upload_path = '/tmp/resized-{}'.format(tmpkey)
    s3_client.download_file(bucket, key, download_path)
    resize_image(download_path, upload_path)
    s3_client.upload_file(upload_path, '{}-resized'.format(bucket), 'resized-
{}'.format(key))
```

2. Dans le même répertoire que celui dans lequel vous avez créé votre fichier `lambda_function.py`, créez un nouveau répertoire nommé `package` et installez la bibliothèque Pillow (PIL) et AWS SDK pour Python (Boto3). Bien que l'exécution Lambda Python inclut une version du kit SDK Boto3, nous vous recommandons d'ajouter toutes les dépendances de votre fonction à votre package de déploiement, même si elles sont incluses dans l'exécution. Pour plus d'informations, consultez la Dépendances d'exécution en Python.

   ```
   mkdir package
   pip install \
   --platform manylinux2014_x86_64 \
   --target=package \
   --implementation cp \
   --python-version 3.12 \
   --only-binary=:all: --upgrade \
   pillow boto3
   ```

   La bibliothèque Pillow contient du code C/C++. En utilisant les options `--platform manylinux_2014_x86_64` et `--only-binary=:all:`, pip téléchargera et installera une version de Pillow contenant des fichiers binaires précompilés compatibles avec le système d'exploitation Amazon Linux 2. Cela garantit que votre package de déploiement fonctionnera dans l'environnement d'exécution Lambda, quels que soient le système d'exploitation et l'architecture de votre ordinateur de création local.

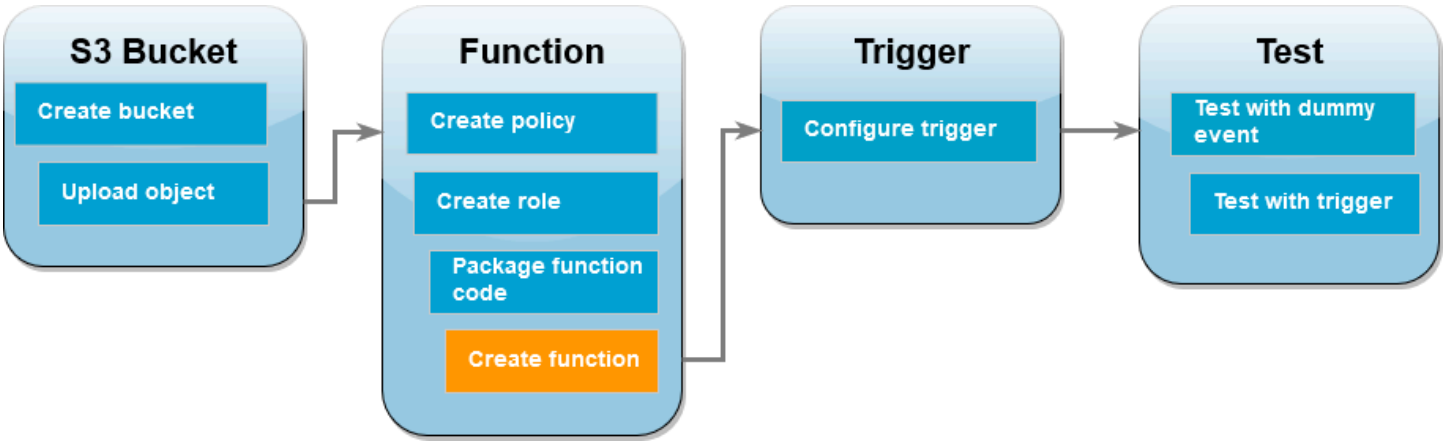
3. Créez un fichier .zip contenant le code de votre application et les bibliothèques Pillow et Boto3. Sous Linux ou macOS, exécutez les commandes suivantes depuis votre interface de ligne de commande.

   ```
   cd package
   zip -r ../lambda_function.zip .
   cd ..
   zip lambda_function.zip lambda_function.py
   ```

   Sous Windows, utilisez l'outil zip de votre choix pour créer le fichier `lambda_function.zip`. Assurez-vous que votre fichier `lambda_function.py` et les dossiers contenant vos dépendances sont installés à la racine du fichier .zip.

Vous pouvez également créer votre package de déploiement à l'aide d'un environnement virtuel Python. Consultez Travailler avec des archives de fichiers .zip pour les fonctions Lambda Python

Créer la fonction Lambda



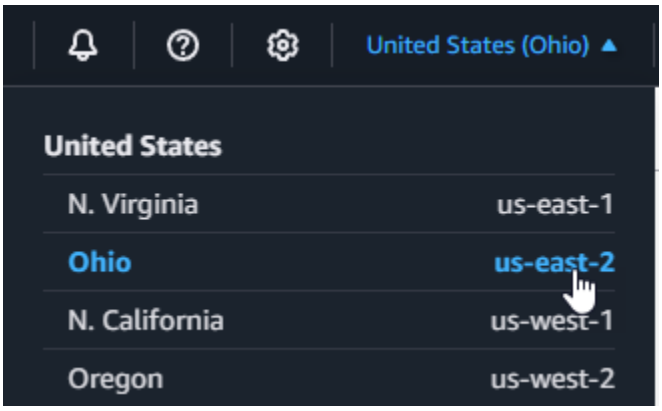
Vous pouvez créer votre fonction Lambda à l'aide de la console Lambda AWS CLI ou de la console Lambda. Suivez les instructions correspondant à la langue de votre choix pour créer la fonction.

AWS Management Console

Pour créer une fonction (console)

Pour créer votre fonction Lambda à l'aide de la console, vous devez d'abord créer une fonction de base contenant du code « Hello world ». Vous remplacez ensuite ce code par votre propre code de fonction en chargeant le fichier .zip ou JAR que vous avez créé à l'étape précédente.

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Assurez-vous de travailler dans le même environnement que celui dans Région AWS lequel vous avez créé votre compartiment Amazon S3. Vous pouvez modifier votre région à l'aide de la liste déroulante en haut de l'écran.



3. Choisissez Créer une fonction.
4. Choisissez Créer à partir de zéro.

5. Sous Informations de base, procédez comme suit :

   a. Sous Nom de la fonction, saisissez **`CreateThumbnail`**.

   b. Pour Environnement d'exécution, choisissez Node.js 22.x ou Python 3.12 en fonction du langage que vous avez choisi pour votre fonction.

   c. Pour Architecture, choisissez x86_64.

6. Dans l'onglet Modifier le rôle d'exécution par défaut, procédez comme suit :

   a. Ouvrez l'onglet, puis choisissez Utiliser un rôle existant.

   b. Sélectionnez le `LambdaS3Role` que vous avez créé précédemment.

7. Choisissez Créer une fonction.

Pour charger le code de fonction (console)

1. Dans le volet Source du code, choisissez Charger à partir de.
2. Choisissez Fichier .zip.
3. Choisissez Charger.
4. Dans le sélecteur de fichiers, sélectionnez votre fichier .zip et choisissez Ouvrir.
5. Choisissez Enregistrer.

AWS CLI

Pour créer la fonction (AWS CLI)

- Exécutez la commande CLI pour la langue que vous avez choisie. Pour le `role` paramètre, assurez-vous de le remplacer `123456789012` par votre propre Compte AWS identifiant. Pour le paramètre `region`, remplacez `us-east-1` par la région dans laquelle vous avez créé vos compartiments Amazon S3.

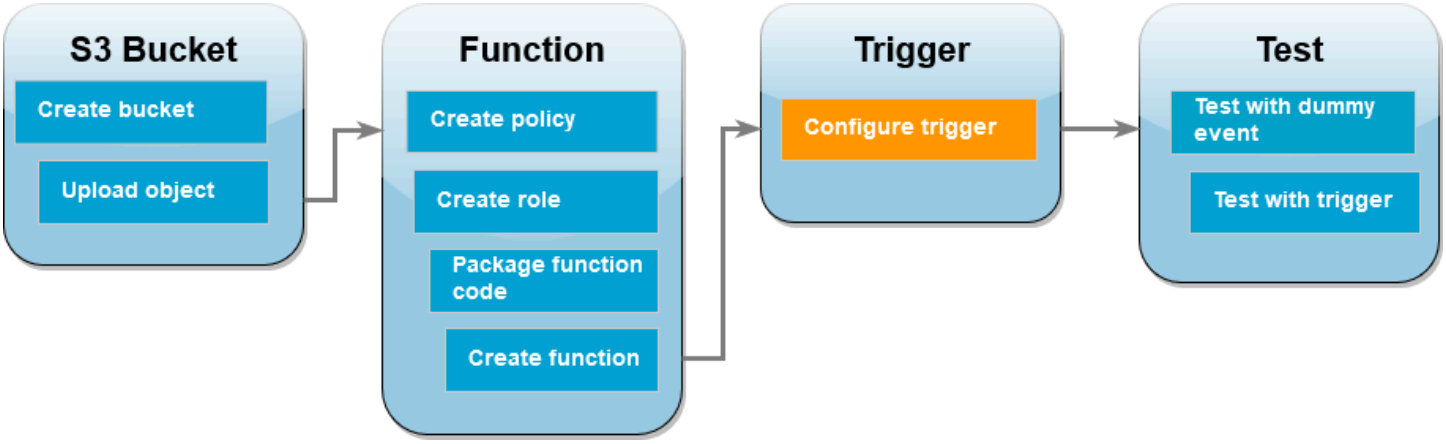
  - Pour Node.js, exécutez la commande suivante depuis le répertoire contenant votre fichier `function.zip`.

```
aws lambda create-function --function-name CreateThumbnail \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs24.x \
--timeout 10 --memory-size 1024 \
--role arn:aws:iam::123456789012:role/LambdaS3Role --region us-east-1
```

- Pour Python, exécutez la commande suivante depuis le répertoire contenant votre fichier `lambda_function.zip`.

```
aws lambda create-function --function-name CreateThumbnail \
--zip-file fileb://lambda_function.zip --handler
 lambda_function.lambda_handler \
--runtime python3.14 --timeout 10 --memory-size 1024 \
--role arn:aws:iam::123456789012:role/LambdaS3Role --region us-east-1
```

Configurer Amazon S3 pour invoquer la fonction



Pour que votre fonction Lambda s'exécute lorsque vous chargez une image dans votre compartiment source, vous devez configurer un déclencheur pour votre fonction. Vous pouvez configurer le déclencheur Amazon S3 à l'aide de la console ou de la AWS CLI.

> **Important**
>
> Cette procédure configure le compartiment Amazon S3 pour qu'il invoque votre fonction chaque fois qu'un objet est créé dans le compartiment. Veillez à configurer cela uniquement pour le compartiment source. Si votre fonction Lambda crée des objets dans le même compartiment que celui qui l'invoque, votre fonction peut être invoquée en continu dans une boucle. Cela peut entraîner la facturation de frais imprévus à votre Compte AWS charge.

AWS Management Console

Pour configurer le déclencheur Amazon S3 (console)

1. Ouvrez la page Fonctions de la console Lambda et choisissez votre fonction (`CreateThumbnail`).
2. Choisissez Add trigger (Ajouter déclencheur).
3. Sélectionnez S3.
4. Sous Compartiment, sélectionnez votre compartiment source.
5. Sous Types d'événements, sélectionnez Tous les événements de création d'objets.
6. Sous Invocation récursive, cochez la case pour confirmer qu'il n'est pas recommandé d'utiliser le même compartiment Amazon S3 pour les entrées et les sorties. Vous pouvez en savoir plus sur les modèles d'invocation récursive dans Lambda en lisant la rubrique Modèles récursifs qui provoquent des fonctions Lambda incontrôlables dans Serverless Land.
7. Choisissez Ajouter.

   Lorsque vous créez un déclencheur à l'aide de la console Lambda, ce dernier crée automatiquement une politique basée sur une ressource pour donner au service que vous sélectionnez l'autorisation d'invoquer votre fonction.

AWS CLI

Pour configurer le déclencheur Amazon S3 (AWS CLI)

1. Pour que votre compartiment source Amazon S3 invoque votre fonction lorsque vous ajoutez un fichier image, vous devez d'abord configurer les autorisations pour votre fonction à l'aide d'une politique basée sur une ressource. Une déclaration de politique basée sur les ressources donne à d'autres personnes Services AWS l'autorisation d'invoquer votre fonction. Pour autoriser Amazon S3 à invoquer votre fonction, exécutez la commande CLI suivante. Assurez-vous de remplacer le `source-account` paramètre par votre propre Compte AWS identifiant et d'utiliser votre propre nom de compartiment source.

```
aws lambda add-permission --function-name CreateThumbnail \
--principal s3.amazonaws.com --statement-id s3invoke --action
 "lambda:InvokeFunction" \
--source-arn arn:aws:s3:::amzn-s3-demo-source-bucket \
--source-account 123456789012
```

La politique que vous définissez à l'aide de cette commande permet à Amazon S3 d'invoquer votre fonction uniquement lorsqu'une action a lieu sur votre compartiment source.

> **Note**
>
> Bien que les noms des compartiments Amazon S3 soient globalement uniques, il est préférable de spécifier que le compartiment doit appartenir à votre compte lorsque vous utilisez des politiques basées sur les ressources. En effet, si vous supprimez un bucket, il est possible qu'un autre Compte AWS en crée un avec le même Amazon Resource Name (ARN).

2. Enregistrez le JSON suivant dans un fichier nommé `notification.json`. Lorsqu'il est appliqué à votre compartiment source, ce JSON configure le compartiment pour qu'il envoie une notification à votre fonction Lambda chaque fois qu'un nouvel objet est ajouté. Remplacez le Compte AWS numéro et, Région AWS dans la fonction Lambda, l'ARN par votre propre numéro de compte et votre propre région.

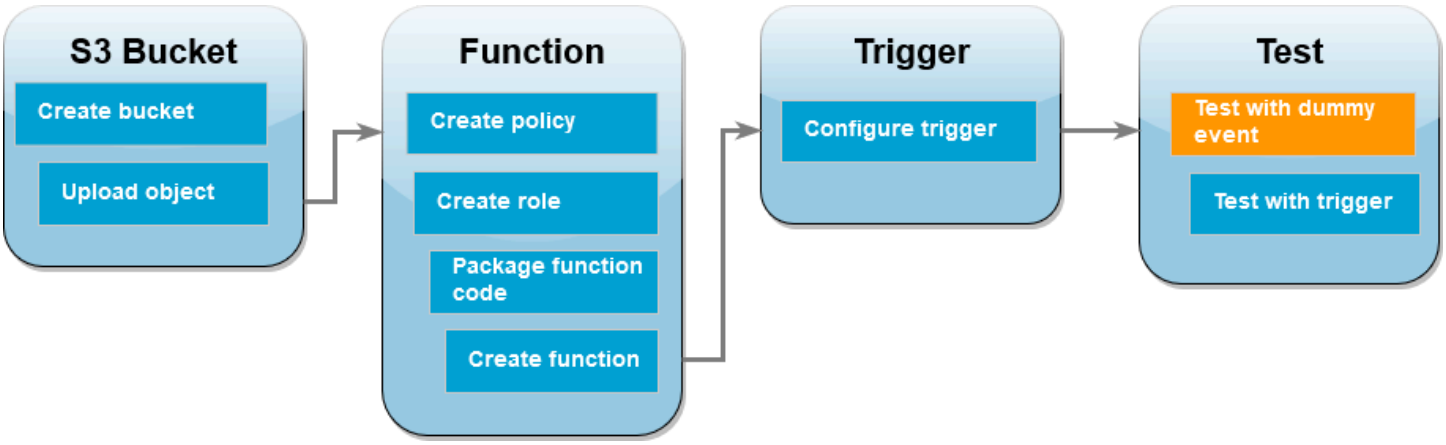
```
{
"LambdaFunctionConfigurations": [
    {
      "Id": "CreateThumbnailEventConfiguration",
      "LambdaFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:CreateThumbnail",
      "Events": [ "s3:ObjectCreated:Put" ]
    }
  ]
}
```

3. Exécutez la commande CLI suivante pour appliquer les paramètres de notification du fichier JSON que vous avez créé à votre compartiment source. Remplacez `amzn-s3-demo-source-bucket` par le nom de votre propre compartiment source.

```
aws s3api put-bucket-notification-configuration --bucket amzn-s3-demo-source-
bucket \
--notification-configuration file://notification.json
```

Pour en savoir plus sur la `put-bucket-notification-configuration` commande et l'`notification-configuration`option, consultez le manuel de référence put-bucket-notification-configurationdes commandes de la AWS CLI.

Test de votre fonction Lambda à l'aide d'un événement fictif



Avant de tester l'ensemble de votre configuration en ajoutant un fichier image à votre compartiment source Amazon S3, vous devez vérifier que votre fonction Lambda fonctionne correctement en l'invoquant avec un événement fictif. Un événement dans Lambda est un document au format JSON qui contient des données à traiter par votre fonction. Lorsque votre fonction est invoquée par Amazon S3, l'événement envoyé à votre fonction contient des informations telles que le nom du compartiment, l'ARN du compartiment et la clé de l'objet.

AWS Management Console

Pour tester votre fonction Lambda à l'aide d'un événement fictif (console)

1. Ouvrez la page Fonctions de la console Lambda et choisissez votre fonction (`CreateThumbnail`).
2. Choisissez l'onglet Test.
3. Pour créer votre événement de test, dans le volet Événement de test, procédez comme suit :
   a. Sous Action d'événement de test, sélectionnez Créer un nouvel événement.
   b. Dans Event name (Nom de l'événement), saisissez **`myTestEvent`**.
   c. Pour Modèle, sélectionnez S3 Put.
   d. Remplacez les valeurs des paramètres suivants par vos propres valeurs.
      - Pour `awsRegion`, remplacez-le `us-east-1` par celui dans Région AWS lequel vous avez créé vos compartiments Amazon S3.
      - Pour `name`, remplacez `amzn-s3-demo-bucket` par le nom de votre propre compartiment source Amazon S3.

- Pour `key`, remplacez `test%2Fkey` par le nom de fichier de l'objet de test que vous avez chargé dans votre compartiment source à l'étape Charger une image de test dans votre compartiment source.

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-1",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "EXAMPLE123456789",
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/
mnopqrstuvwxyzABCDEFGH"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "amzn-s3-demo-bucket",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          },
          "arn": "arn:aws:s3:::amzn-s3-demo-bucket"
        },
        "object": {
          "key": "test%2Fkey",
          "size": 1024,
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901"
        }
      }
    }
  ]
```

```
}
```

e. Choisissez Enregistrer.

4. Dans le volet Événement de test, choisissez Test.
5. Pour vérifier que votre fonction a créé une version redimensionnée de votre image et l'a stockée dans votre compartiment Amazon S3 cible, procédez comme suit :

   a. Ouvrez la page Compartiments de la console Amazon S3.

   b. Choisissez votre compartiment cible et vérifiez que votre fichier redimensionné est répertorié dans le volet Objets.

AWS CLI

Pour tester de votre fonction Lambda à l'aide d'un événement fictif (AWS CLI)

1. Enregistrez le JSON suivant dans un fichier nommé `dummyS3Event.json`. Remplacez les valeurs des paramètres suivants par vos propres valeurs :

   - Pour `awsRegion`, remplacez-le `us-east-1` par celui dans Région AWS lequel vous avez créé vos compartiments Amazon S3.
   - Pour `name`, remplacez `amzn-s3-demo-bucket` par le nom de votre propre compartiment source Amazon S3.
   - Pour `key`, remplacez `test%2Fkey` par le nom de fichier de l'objet de test que vous avez chargé dans votre compartiment source à l'étape Charger une image de test dans votre compartiment source.

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-1",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
      "requestParameters": {
```

```
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "EXAMPLE123456789",
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/
mnopqrstuvwxyzABCDEFGH"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "amzn-s3-demo-bucket",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          },
          "arn": "arn:aws:s3:::amzn-s3-demo-bucket"
        },
        "object": {
          "key": "test%2Fkey",
          "size": 1024,
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901"
        }
      }
    }
  ]
}
```

2. À partir du répertoire dans lequel vous avez enregistré votre fichier dummyS3Event.json, invoquez la fonction en exécutant la commande CLI suivante. Cette commande invoque votre fonction Lambda de manière synchrone en la spécifiant RequestResponse comme valeur du paramètre de type d'invocation. Pour en savoir plus sur l'invocation synchrone et asynchrone, consultez Invocation de fonctions Lambda.

```
aws lambda invoke --function-name CreateThumbnail \
--invocation-type RequestResponse --cli-binary-format raw-in-base64-out \
--payload file://dummyS3Event.json outputfile.txt
```

cli-binary-formatCette option est obligatoire si vous utilisez la version 2 du AWS CLI. Pour faire de ce paramètre le paramètre par défaut, exécutez aws configure set cli-binary-format raw-in-base64-out. Pour plus d'informations, veuillez consulter les AWS CLI options de ligne de commande.

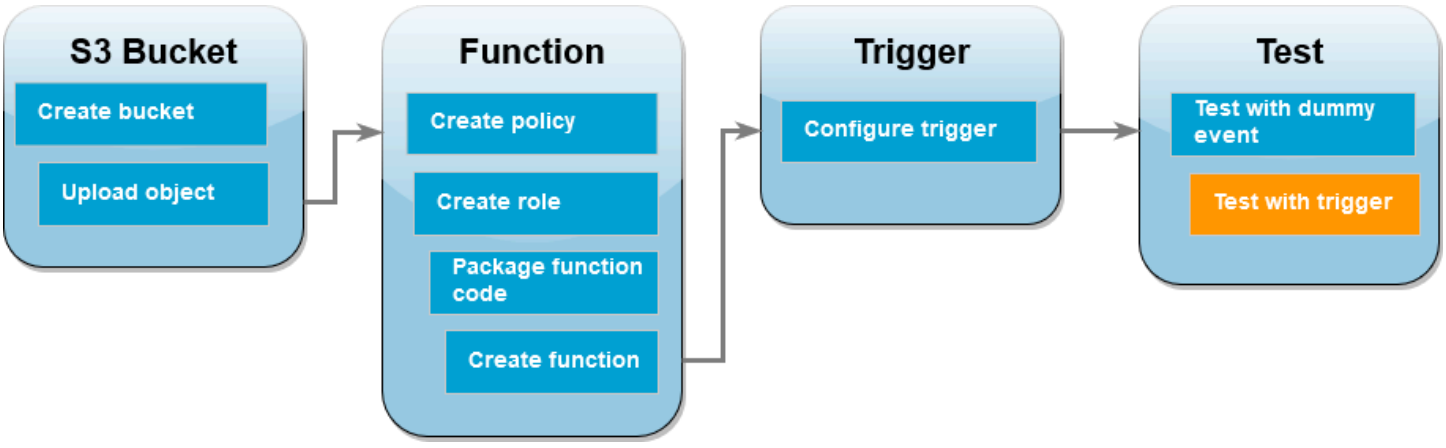
3. Vérifiez que votre fonction a créé une version miniature de votre image et l'a enregistrée dans votre compartiment Amazon S3 cible. Exécutez la commande CLI suivante, en remplaçant `amzn-s3-demo-source-bucket-resized` par le nom de votre compartiment de destination.

```
aws s3api list-objects-v2 --bucket amzn-s3-demo-source-bucket-resized
```

Vous devez visualiser des résultats similaires à ce qui suit. Le paramètre `Key` indique le nom de fichier de votre fichier image redimensionné.

```
{
    "Contents": [
        {
            "Key": "resized-HappyFace.jpg",
            "LastModified": "2023-06-06T21:40:07+00:00",
            "ETag": "\"d8ca652ffe83ba6b721ffc20d9d7174a\"",
            "Size": 2633,
            "StorageClass": "STANDARD"
        }
    ]
}
```

Tester votre fonction à l'aide du déclencheur Amazon S3



Maintenant que vous avez confirmé que votre fonction Lambda fonctionne correctement, vous êtes prêt à tester votre configuration complète en ajoutant un fichier image à votre compartiment source Amazon S3. Lorsque vous ajoutez votre image au compartiment source, votre fonction Lambda doit

être automatiquement invoquée. Votre fonction crée une version redimensionnée du fichier et la stocke dans votre compartiment cible.

AWS Management Console

Pour tester votre fonction Lambda à l'aide du déclencheur Amazon S3 (console)

1. Pour charger une image dans votre compartiment Amazon S3, procédez comme suit :
   a. Ouvrez la page Compartiments de la console Amazon S3 et choisissez votre compartiment source.
   b. Choisissez Charger.
   c. Choisissez Ajouter des fichiers et utilisez le sélecteur de fichiers pour sélectionner le fichier image que vous souhaitez charger. Votre objet image peut être n'importe quel fichier .jpg ou .png.
   d. Choisissez Ouvrir, puis Charger.
2. Vérifiez que Lambda a enregistré une version redimensionnée de votre fichier image dans votre compartiment cible en procédant comme suit :
   a. Retournez à la page Compartiments de la console Amazon S3 et choisissez votre compartiment de destination.
   b. Dans le volet Objets, vous devriez maintenant voir deux fichiers image redimensionnés, un pour chaque test de votre fonction Lambda. Pour télécharger votre image redimensionnée, sélectionnez le fichier, puis choisissez Télécharger.

AWS CLI

Pour tester votre fonction Lambda à l'aide du déclencheur Amazon S3 (AWS CLI)

1. À partir du répertoire contenant l'image que vous souhaitez charger, exécutez la commande CLI suivante. Remplacez le paramètre `--bucket` par le nom de votre compartiment source. Pour les paramètres `--key` et `--body`, utilisez le nom de fichier de votre image de test. Votre image de test peut être n'importe quel fichier .jpg ou .png.

```
aws s3api put-object --bucket amzn-s3-demo-source-bucket --key SmileyFace.jpg --
body ./SmileyFace.jpg
```

2. Vérifiez que votre fonction a créé une version miniature de votre image et l'a enregistrée dans votre compartiment Amazon S3 cible. Exécutez la commande CLI suivante, en remplaçant `amzn-s3-demo-source-bucket-resized` par le nom de votre compartiment de destination.

```
aws s3api list-objects-v2 --bucket amzn-s3-demo-source-bucket-resized
```

Si votre fonction s'exécute correctement, vous obtiendrez un résultat similaire à celui qui suit. Votre compartiment cible doit désormais contenir deux fichiers redimensionnés.

```
{
    "Contents": [
        {
            "Key": "resized-HappyFace.jpg",
            "LastModified": "2023-06-07T00:15:50+00:00",
            "ETag": "\"7781a43e765a8301713f533d70968a1e\"",
            "Size": 2763,
            "StorageClass": "STANDARD"
        },
        {
            "Key": "resized-SmileyFace.jpg",
            "LastModified": "2023-06-07T00:13:18+00:00",
            "ETag": "\"ca536e5a1b9e32b22cd549e18792cdbc\"",
            "Size": 1245,
            "StorageClass": "STANDARD"
        }
    ]
}
```

Nettoyage de vos ressources

Vous pouvez maintenant supprimer les ressources que vous avez créées pour ce didacticiel, sauf si vous souhaitez les conserver. En supprimant AWS les ressources que vous n'utilisez plus, vous évitez des frais inutiles pour votre Compte AWS.

Pour supprimer la fonction Lambda

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction que vous avez créée.

3. Sélectionnez Actions, Supprimer.
4. Saisissez **`confirm`** dans la zone de saisie de texte et choisissez Delete (Supprimer).

Suppression de la stratégie que vous avez créée

1. Ouvrez la page Policies (Stratégies) de la console IAM.
2. Sélectionnez la politique que vous avez créée (AWSLambdaS3Policy).
3. Choisissez Actions de stratégie, Supprimer.
4. Sélectionnez Delete (Supprimer).

Pour supprimer le rôle d'exécution

1. Ouvrez la page Roles (Rôles) de la console IAM.
2. Sélectionnez le rôle d'exécution que vous avez créé.
3. Sélectionnez Delete (Supprimer).
4. Saisissez le nom du rôle dans le champ de saisie de texte et choisissez Delete (Supprimer).

Pour supprimer le compartiment S3

1. Ouvrez la console Amazon S3.
2. Sélectionnez le compartiment que vous avez créé.
3. Sélectionnez Delete (Supprimer).
4. Saisissez le nom du compartiment dans le champ de saisie de texte.
5. Choisissez Supprimer le compartiment.

Utiliser des secrets Secrets Manager dans des fonctions Lambda

AWS Secrets Manager vous aide à gérer les informations d'identification, les clés d'API et les autres secrets dont vos fonctions Lambda ont besoin. Vous disposez de deux approches principales pour récupérer des secrets dans vos fonctions Lambda, toutes deux offrant de meilleures performances et des coûts réduits par rapport à la récupération de secrets directement à l'aide du SDK : AWS

- AWS paramètres et secrets Extension Lambda : solution indépendante de l'exécution qui fournit une interface HTTP simple pour récupérer les secrets
- Utilitaire Powertools for AWS Lambda Parameters : solution intégrée au code qui prend en charge plusieurs fournisseurs (Secrets Manager, Parameter Store AppConfig) avec des transformations intégrées

Les deux approches conservent des caches locaux de secrets, ce qui évite à votre fonction d’appeler Secrets Manager pour chaque invocation. Lorsque votre fonction demande un secret, le cache est d’abord vérifié. Si le secret est disponible et n’a pas expiré, il est renvoyé immédiatement. Sinon, il est extrait de Secrets Manager, mis en cache et renvoyé. Ce mécanisme de mise en cache permet d’accélérer les temps de réponse et de réduire les coûts en minimisant les appels d’API.

Choix d’une approche

Tenez compte des facteurs suivants lorsque vous choisissez entre l'extension et PowerTools :

Utilisez l'extension Lambda de AWS paramètres et de secrets lorsque :

- Vous recherchez une solution indépendante de l’environnement d’exécution qui fonctionne avec n’importe quel environnement d’exécution Lambda
- Vous préférez ne pas ajouter de dépendances de code à votre fonction
- Vous souhaitez seulement récupérer des secrets à partir de Secrets Manager ou de Parameter Store

Utilisez Powertools pour l'utilitaire AWS Lambda de paramètres lorsque :

- Vous souhaitez bénéficier d’une expérience de développement intégrée au code de votre application
- Vous avez besoin d'assistance pour plusieurs fournisseurs (Secrets Manager, Parameter Store, AppConfig)
- Vous souhaitez des transformations de données intégrées (analyse JSON, décodage Base64)
- Vous utilisez des environnements d'exécution Python TypeScript, Java ou .NET

Quand utiliser Secrets Manager avec Lambda

Les scénarios courants d'utilisation de Secrets Manager avec Lambda comprennent les suivants :

- Stockage d'informations d'identification de base de données que votre fonction utilise pour se connecter à Amazon RDS ou d'autres bases de données
- Gestion des clés d'API pour les services externes appelés par votre fonction
- Stockage des clés de chiffrement ou d'autres données de configuration sensibles
- Rotation automatique des informations d'identification sans qu'il soit nécessaire de mettre à jour le code de votre fonction

Utilisation des AWS paramètres et des secrets de l'extension Lambda

L'extension Lambda de AWS paramètres et de secrets utilise une interface HTTP simple compatible avec n'importe quel environnement d'exécution Lambda. Par défaut, elle met en cache les secrets pendant 300 secondes (5 minutes) et peut contenir jusqu'à 1 000 secrets. Vous pouvez personnaliser ces paramètres à l'aide de variables d'environnement.

Utiliser Secrets Manager dans une fonction Lambda

Cette section part du principe que vous possédez déjà un secret Secrets Manager. Pour créer un secret, voir Création d'un AWS Secrets Manager secret.

Création du package de déploiement

Choisissez votre environnement d'exécution préféré et suivez les étapes pour créer une fonction qui récupère les secrets depuis Secrets Manager. L'exemple de fonction extrait un secret dans Secrets Manager et peut être utilisé pour accéder aux informations d'identification de base de données, aux clés d'API ou à d'autres données de configuration sensibles dans vos applications.

Python

Pour créer une fonction Python

1. Créez et accédez à un nouveau répertoire de projet. Exemple :

```
mkdir my_function
cd my_function
```

2. Créez un fichier nommé `lambda_function.py` avec le code suivant. Pour `secret_name`, utilisez le nom ou l'Amazon Resource Name (ARN) de votre secret.

```
import json
import os
import requests

def lambda_handler(event, context):
    try:
        # Replace with the name or ARN of your secret
        secret_name = "arn:aws:secretsmanager:us-
east-1:111122223333:secret:SECRET_NAME"

        secrets_extension_endpoint = f"http://localhost:2773/secretsmanager/get?
secretId={secret_name}"
        headers = {"X-Aws-Parameters-Secrets-Token":
 os.environ.get('AWS_SESSION_TOKEN')}

        response = requests.get(secrets_extension_endpoint, headers=headers)
        print(f"Response status code: {response.status_code}")

        secret = json.loads(response.text)["SecretString"]
        print(f"Retrieved secret: {secret}")

        return {
            'statusCode': response.status_code,
            'body': json.dumps({
                'message': 'Successfully retrieved secret',
                'secretRetrieved': True
            })
        }

    except Exception as e:
        print(f"Error: {str(e)}")
        return {
            'statusCode': 500,
            'body': json.dumps({
                'message': 'Error retrieving secret',
                'error': str(e)
            })
        }
```

3. Créez un fichier nommé `requirements.txt` avec ce contenu :

```
requests
```

4. Installez les dépendances :

```
pip install -r requirements.txt -t .
```

5. Créez un fichier .zip contenant tous les fichiers :

```
zip -r function.zip .
```

Node.js

Pour créer une fonction Node.js.

1. Créez et accédez à un nouveau répertoire de projet. Exemple :

```
mkdir my_function
cd my_function
```

2. Créez un fichier nommé `index.mjs` avec le code suivant. Pour `secret_name`, utilisez le nom ou l'Amazon Resource Name (ARN) de votre secret.

```
import http from 'http';

export const handler = async (event) => {
    try {
        // Replace with the name or ARN of your secret
        const secretName = "arn:aws:secretsmanager:us-
east-1:111122223333:secret:SECRET_NAME";
        const options = {
            hostname: 'localhost',
            port: 2773,
            path: `/secretsmanager/get?secretId=${secretName}`,
            headers: {
                'X-Aws-Parameters-Secrets-Token': process.env.AWS_SESSION_TOKEN
            }
        };

        const response = await new Promise((resolve, reject) => {
            http.get(options, (res) => {
```

```
                let data = '';
                res.on('data', (chunk) => { data += chunk; });
                res.on('end', () => {
                    resolve({
                        statusCode: res.statusCode,
                        body: data
                    });
                });
            }).on('error', reject);
        });

        const secret = JSON.parse(response.body).SecretString;
        console.log('Retrieved secret:', secret);

        return {
            statusCode: response.statusCode,
            body: JSON.stringify({
                message: 'Successfully retrieved secret',
                secretRetrieved: true
            })
        };
    } catch (error) {
        console.error('Error:', error);
        return {
            statusCode: 500,
            body: JSON.stringify({
                message: 'Error retrieving secret',
                error: error.message
            })
        };
    }
};
```

3. Créez un fichier .zip contient le fichier `index.mjs`.

```
zip -r function.zip index.mjs
```

Java

Pour créer une fonction Java

1. Créez un projet Maven :

```
mvn archetype:generate \
    -DgroupId=example \
    -DartifactId=lambda-secrets-demo \
    -DarchetypeArtifactId=maven-archetype-quickstart \
    -DarchetypeVersion=1.4 \
    -DinteractiveMode=false
```

2. Accédez au répertoire du projet :

```
cd lambda-secrets-demo
```

3. Ouvrez `pom.xml` et remplacez le contenu par ce qui suit :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>example</groupId>
    <artifactId>lambda-secrets-demo</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-core</artifactId>
            <version>1.2.1</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>3.2.4</version>
```

```
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            <createDependencyReducedPom>false</
createDependencyReducedPom>
                            <finalName>function</finalName>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

4. Renommez `/lambda-secrets-demo/src/main/java/example/App.java` en `Hello.java` pour qu'il corresponde au nom du gestionnaire Java par défaut de Lambda (`example.Hello::handleRequest`) :

```
mv src/main/java/example/App.java src/main/java/example/Hello.java
```

5. Ouvrez le fichier `Hello.java` et remplacez son contenu par ce qui suit. Pour `secretName`, utilisez le nom ou l'Amazon Resource Name (ARN) de votre secret.

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;

public class Hello implements RequestHandler<Object, String> {
    private final HttpClient client = HttpClient.newHttpClient();

    @Override
    public String handleRequest(Object input, Context context) {
        try {
            // Replace with the name or ARN of your secret
```

```
            String secretName = "arn:aws:secretsmanager:us-
east-1:111122223333:secret:SECRET_NAME";
            String endpoint = "http://localhost:2773/secretsmanager/get?
secretId=" + secretName;

            HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create(endpoint))
                .header("X-Aws-Parameters-Secrets-Token",
 System.getenv("AWS_SESSION_TOKEN"))
                .GET()
                .build();

            HttpResponse<String> response = client.send(request,
                HttpResponse.BodyHandlers.ofString());

            String secret = response.body();
            secret = secret.substring(secret.indexOf("SecretString") + 15);
            secret = secret.substring(0, secret.indexOf("\""));

            System.out.println("Retrieved secret: " + secret);
            return String.format(
                "{\"statusCode\": %d, \"body\": \"%s\"}",
                response.statusCode(), "Successfully retrieved secret"
            );

        } catch (Exception e) {
            e.printStackTrace();
            return String.format(
                "{\"body\": \"Error retrieving secret: %s\"}",
                e.getMessage()
            );
        }
    }
}
```

6. Supprimez le répertoire de test. Maven le crée par défaut, mais nous n'en avons pas besoin pour cet exemple.

```
rm -rf src/test
```

7. Générez le projet :

```
mvn package
```

8. Téléchargez le fichier JAR (`target/function.jar`) pour une utilisation ultérieure.

Créer la fonction

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez Créer une fonction.
3. Sélectionnez Créer à partir de zéro.
4. Sous Nom de la fonction, saisissez **`secret-retrieval-demo`**.
5. Choisissez votre Environnement d'exécution préféré.
6. Choisissez Créer une fonction.

Charger un package de déploiement

1. Dans l'onglet Code de la fonction, choisissez Charger à partir de et sélectionnez Fichier .zip (pour Python et Node.js) ou Fichier .jar (pour Java).
2. Chargez le package de déploiement créé précédemment.
3. Choisissez Enregistrer.

Ajouter l'extension

Pour ajouter l'extension Lambda AWS Parameters and Secrets en tant que couche

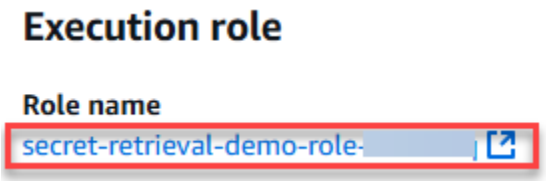
1. Dans l'onglet Code de la fonction, faites défiler la page jusqu'à Couches.
2. Choisissez Add a layer (Ajouter une couche).
3. Sélectionnez Couches AWS .
4. Choisissez AWS-Parameters-and-Secrets-Lambda-Extension.
5. Sélectionnez la dernière version.
6. Choisissez Ajouter.

Ajout d'autorisations

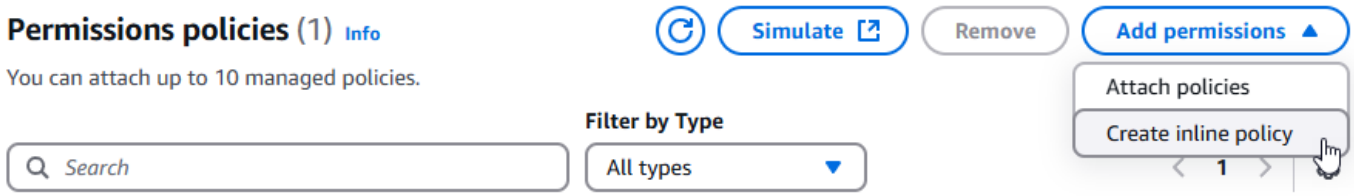
Ajouter des autorisations Secrets Manager à votre rôle d'exécution

1. Choisissez l'onglet Configuration, puis Permissions (Autorisations).

2. Sous Nom du rôle, cliquez sur le lien vers votre rôle d'exécution. Ce lien ouvre le rôle dans la console IAM.



3. Sélectionnez Ajouter des autorisations, puis Ajouter la politique.



4. Choisissez l'onglet JSON et ajoutez la politique suivante. Pour `Resource`, saisissez l'ARN de votre secret.

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "secretsmanager:GetSecretValue",
            "Resource": "arn:aws:secretsmanager:us-
east-1:111122223333:secret:SECRET_NAME"
        }
    ]
}
```

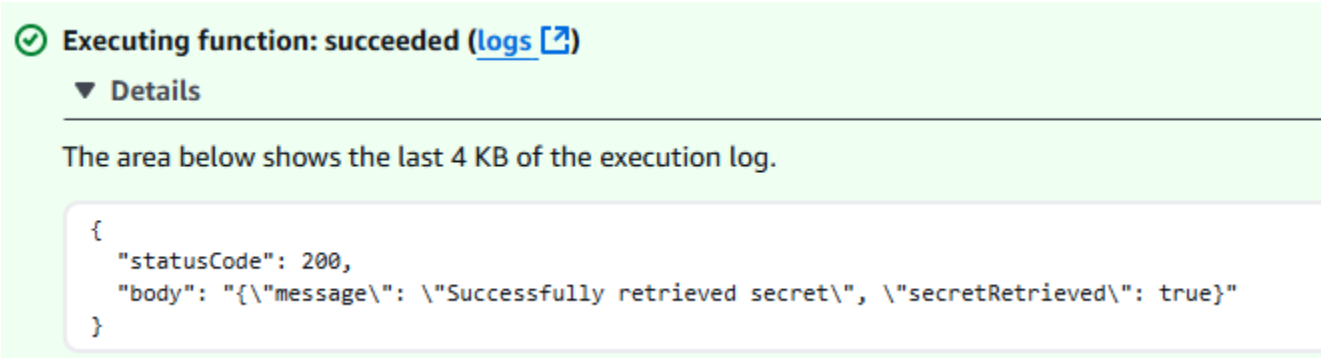
5. Choisissez Suivant.
6. Entrez le nom de la politique.
7. Choisissez Create Policy (Créer une politique).

Tester la fonction

Pour tester la fonction

1. Retournez à la console Lambda.

2. Sélectionnez l'onglet Test.
3. Sélectionnez Tester). Vous devriez voir la réponse suivante :



Variables d'environnement

L'extension Lambda AWS Parameters and Secrets utilise les paramètres par défaut suivants. Vous pouvez remplacer ces paramètres en créant les variables d'environnement correspondantes. Pour afficher les paramètres actuels d'une fonction, réglez `PARAMETERS_SECRETS_EXTENSION_LOG_LEVEL` sur `DEBUG`. L'extension enregistre ses informations de configuration dans CloudWatch Logs au début de chaque appel de fonction.

Paramètre	Valeur par défaut	Valeurs valides	Variable d'environnement	Détails
Port HTTP	2773	1 – 65535	PARAMETERS_SECRETS_EXTENSION_HTTP_PORT	Port du serveur HTTP local
Cache activé	TRUE	TRUE | FALSE	PARAMETERS_SECRETS_EXTENSION_CACHE_ENABLED	Activer ou désactiver le cache
Taille du cache	1 000	0 - 1000	PARAMETERS_SECRETS_EXTENSION_CACHE_SIZE	Définissez cette valeur sur 0 pour désactiver la mise en cache
TTL de Secrets Manager	300 seconde	0 à 300 seconde	SECRETS_MANAGER_TTL	Time-to-live pour les secrets mis en cache. Définissez sur 0 pour

Paramètre	Valeur par défaut	Valeurs valides	Variable d'environnement	Détails
				désactiver la mise en cache. Cette variable est ignorée si la valeur de PARAMETER S_SECRETS_EXTENSIO N_CACHE_SIZE est de 0.
TTL du magasin de paramètres	300 seconde	0 à 300 seconde	SSM_PARAM ETER_STORE_TTL	Time-to-live pour les paramètres mis en cache. Définissez sur 0 pour désactiver la mise en cache. Cette variable est ignorée si la valeur de PARAMETER S_SECRETS_EXTENSIO N_CACHE_SIZE est de 0.
Niveau du journal	INFO	DEBUG | INFO | WARN | ERROR | NONE	PARAMETER S_SECRETS_EXTENSIO N_LOG_LEVEL	Le niveau de détail indiqué dans les journaux pour l'extension
Nombre maximal de connexions	3	1 ou plus	PARAMETER S_SECRETS_EXTENSIO N_MAX_CONNECTIONS	Nombre maximum de connexions HTTP pour les requêtes à Parameter Store ou Secrets Manager.
Délai d'expiration de Secrets Manager	0 (n'expire pas)	Tous les nombres entiers	SECRETS_M ANAGER_TIMEOUT_MIL LIS	Délai d'expiration des requêtes adressées à Secrets Manager (en millisecondes)

Paramètre	Valeur par défaut	Valeurs valides	Variable d'environnement	Détails
Délai d'expiration de Parameter Store	0 (n'expire pas)	Tous les nombres entiers	SSM_PARAMETER_STORE_TIMEOUT_MILLIS	Délai d'expiration des requêtes adressées au Parameter Store (en millisecondes)

Utilisation d'une rotation des secrets

Si vous alternez fréquemment les secrets, la durée de cache par défaut de 300 secondes peut entraîner l'utilisation de secrets obsolètes par votre fonction. Deux options s'offrent à vous pour vous assurer que votre fonction utilise la dernière valeur de secret :

- Réduisez le TTL du cache en définissant la variable d'environnement `SECRETS_MANAGER_TTL` sur une valeur inférieure (en secondes). Par exemple, configurez-le sur `60` pour que votre fonction n'utilise jamais un secret vieux de plus d'une minute.
- Utilisez les étiquettes intermédiaires `AWSCURRENT` ou `AWSPREVIOUS` dans votre demande de secret pour vous assurer d'obtenir la version spécifique que vous souhaitez :

```
secretsmanager/get?secretId=YOUR_SECRET_NAME&versionStage=AWSCURRENT
```

Choisissez l'approche qui équilibre le mieux vos besoins en matière de performances et de fraîcheur. Un TTL inférieur signifie des appels plus fréquents à Secrets Manager, mais garantit que vous travaillez avec les valeurs de secret les plus récentes.

Utilisation de l'utilitaire de paramètres de Powertools pour AWS Lambda

L'utilitaire de paramètres de Powertools for AWS Lambda fournit une interface unifiée permettant de récupérer les secrets auprès de plusieurs fournisseurs, notamment Secrets Manager, le magasin de paramètres et. AppConfig Il gère la mise en cache, les transformations et fournit une expérience de développement plus intégrée par rapport à l'approche de l'extension.

Avantages de l’utilitaire de paramètres

- Fournisseurs multiples - Récupérez les paramètres depuis Secrets Manager, Parameter Store et AppConfig en utilisant la même interface
- Transformations intégrées : analyse automatique du JSON, décodage Base64 et autres transformations de données
- Mise en cache intégrée : mise en cache configurable avec prise en charge de TTL pour réduire les appels d’API
- Sécurité typographique : prise en charge efficace de la saisie dans les environnements d'exécution compatibles TypeScript et dans les autres environnements d'exécution compatibles
- Gestion des erreurs : logique de nouvelle tentative et gestion des erreurs intégrées

Exemples de code

Les exemples suivants montrent comment récupérer des secrets à l’aide de l’utilitaire de paramètres dans différents environnements d’exécution :

Python

> **Note**
>
> Pour obtenir des exemples complets et des instructions de configuration, consultez la documentation de l’utilitaire de paramètres.

Récupération de secrets depuis Secrets Manager à l'aide de l'utilitaire Powertools for AWS Lambda Parameters.

```
from aws_lambda_powertools import Logger
from aws_lambda_powertools.utilities import parameters

logger = Logger()

def lambda_handler(event, context):
    try:
        # Get secret with caching (default TTL: 5 seconds)
        secret_value = parameters.get_secret("my-secret-name")

        # Get secret with custom TTL
```

```
        secret_with_ttl = parameters.get_secret("my-secret-name", max_age=300)

        # Get secret and transform JSON
        secret_json = parameters.get_secret("my-json-secret", transform="json")

        logger.info("Successfully retrieved secrets")

        return {
            'statusCode': 200,
            'body': 'Successfully retrieved secrets'
        }

    except Exception as e:
        logger.error(f"Error retrieving secret: {str(e)}")
        return {
            'statusCode': 500,
            'body': f'Error: {str(e)}'
        }
```

TypeScript

> **Note**
>
> Pour obtenir des exemples complets et des instructions de configuration, consultez la documentation de l'utilitaire de paramètres.

Récupération de secrets depuis Secrets Manager à l'aide de l'utilitaire Powertools for AWS Lambda Parameters.

```
import { Logger } from '@aws-lambda-powertools/logger';
import { getSecret } from '@aws-lambda-powertools/parameters/secrets';
import type { Context } from 'aws-lambda';

const logger = new Logger();

export const handler = async (event: any, context: Context) => {
    try {
        // Get secret with caching (default TTL: 5 seconds)
        const secretValue = await getSecret('my-secret-name');

        // Get secret with custom TTL
```

```
        const secretWithTtl = await getSecret('my-secret-name', { maxAge: 300 });

        // Get secret and transform JSON
        const secretJson = await getSecret('my-json-secret', { transform: 'json' });

        logger.info('Successfully retrieved secrets');

        return {
            statusCode: 200,
            body: 'Successfully retrieved secrets'
        };

    } catch (error) {
        logger.error('Error retrieving secret', { error });
        return {
            statusCode: 500,
            body: `Error: ${error}`
        };
    }
};
```

Java

> **Note**
>
> Pour obtenir des exemples complets et des instructions de configuration, consultez la documentation de l'utilitaire de paramètres.

Récupération de secrets depuis Secrets Manager à l'aide de l'utilitaire Powertools for AWS Lambda Parameters.

```
import software.amazon.lambda.powertools.logging.Logging;
import software.amazon.lambda.powertools.parameters.SecretsProvider;
import software.amazon.lambda.powertools.parameters.ParamManager;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

public class SecretHandler implements RequestHandler<Object, String> {

    private final SecretsProvider secretsProvider =
 ParamManager.getSecretsProvider();
```

```
    @Logging
    @Override
    public String handleRequest(Object input, Context context) {
        try {
            // Get secret with caching (default TTL: 5 seconds)
            String secretValue = secretsProvider.get("my-secret-name");

            // Get secret with custom TTL (300 seconds)
            String secretWithTtl = secretsProvider.withMaxAge(300).get("my-secret-
name");

            // Get secret and transform JSON
            MySecret secretJson = secretsProvider.get("my-json-secret",
 MySecret.class);

            return "Successfully retrieved secrets";

        } catch (Exception e) {
            return "Error retrieving secret: " + e.getMessage();
        }
    }

    public static class MySecret {
        // Define your secret structure here
    }
}
```

.NET

> **Note**
>
> Pour obtenir des exemples complets et des instructions de configuration, consultez la documentation de l'utilitaire de paramètres.

Récupération de secrets depuis Secrets Manager à l'aide de l'utilitaire Powertools for AWS Lambda Parameters.

```
using AWS.Lambda.Powertools.Logging;
using AWS.Lambda.Powertools.Parameters;
using Amazon.Lambda.Core;
```

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeriali

public class Function
{
    private readonly ISecretsProvider _secretsProvider;

    public Function()
    {
        _secretsProvider = ParametersManager.SecretsProvider;
    }

    [Logging]
    public async Task<string> FunctionHandler(object input, ILambdaContext context)
    {
        try
        {
            // Get secret with caching (default TTL: 5 seconds)
            var secretValue = await _secretsProvider.GetAsync("my-secret-name");

            // Get secret with custom TTL
            var secretWithTtl = await
 _secretsProvider.WithMaxAge(TimeSpan.FromMinutes(5))
                .GetAsync("my-secret-name");

            // Get secret and transform JSON
            var secretJson = await _secretsProvider.GetAsync<MySecret>("my-json-
secret");

            return "Successfully retrieved secrets";
        }
        catch (Exception e)
        {
            return $"Error retrieving secret: {e.Message}";
        }
    }

    public class MySecret
    {
        // Define your secret structure here
    }
}
```

Configuration et autorisations

Pour utiliser l'utilitaire de paramètres :

1. Installez Powertools AWS Lambda pour votre environnement d'exécution. Pour en savoir plus, consultez Powertools pour AWS Lambda.
2. Ajoutez les autorisations IAM nécessaires au rôle d'exécution de votre fonction. Pour plus d'informations, consultez Gestion des autorisations dans AWS Lambda.
3. Configurez tous les paramètres facultatifs par le biais de variables d'environnement.

Les autorisations IAM requises sont les mêmes que pour l'approche de l'extension. L'utilitaire gère automatiquement la mise en cache et les appels d'API à Secrets Manager en fonction de votre configuration.

Utilisation de Lambda avec Amazon SQS

> **Note**
>
> Si vous souhaitez envoyer des données à une cible autre qu'une fonction Lambda ou enrichir les données avant de les envoyer, consultez Amazon EventBridge Pipes.

Vous pouvez utiliser une fonction Lambda pour traiter les messages figurant dans une file d'attente Amazon Simple Queue Service (Amazon SQS). Lambda prend en charge à la fois les files d'attente standard et les files d'attente FIFO (premier entré, premier sorti) pour les mappages des sources d'événements. Vous pouvez également utiliser le mode provisionné pour allouer des ressources d'interrogation dédiées pour vos mappages de sources d'événements Amazon SQS. La fonction Lambda et la file d'attente Amazon SQS doivent être Région AWS identiques, bien qu'elles puissent être différentes. Comptes AWS

Lorsque vous traitez des messages Amazon SQS, vous devez implémenter une logique de réponse partielle de lot afin d'éviter que les messages traités avec succès soient de nouveau essayés en cas d'échec de certains messages d'un lot. L'utilitaire Batch Processor de Powertools AWS Lambda simplifie cette implémentation en gérant automatiquement la logique de réponse partielle par lots, en réduisant le temps de développement et en améliorant la fiabilité.

Rubriques

- Comprendre le comportement d'interrogation et de traitement par lots pour les mappages des sources d'événements Amazon SQS
- Utilisation du mode provisionné avec les mappages de sources d'événements Amazon SQS
- Configuration du mode provisionné pour le mappage des sources d'événements Amazon SQS
- Exemple d'événement de message de file d'attente standard
- Exemple d'événement de message de file d'attente FIFO
- Création et configuration d'un mappage des sources d'événements Amazon SQS
- Configuration du comportement de mise à l'échelle pour les mappages des sources d'événements SQS
- Gestion des erreurs pour une source d'événements SQS dans Lambda
- Paramètres Lambda pour les mappages des sources d'événement Amazon SQS
- Utilisation du filtrage des événements avec une source d'événement Amazon SQS

- Didacticiel : utilisation de Lambda avec Amazon SQS
- Didacticiel : utilisation d'une file d'attente Amazon SQS entre comptes en tant que source d'événement

Comprendre le comportement d'interrogation et de traitement par lots pour les mappages des sources d'événements Amazon SQS

Avec les mappages des sources d'événements Amazon SQS, Lambda interroge la file d'attente et invoque votre fonction de manière synchrone avec un événement. Chaque événement peut contenir un lot de plusieurs messages provenant de la file d'attente. Lambda reçoit ces événements un lot à la fois, et invoque votre fonction une fois pour chaque lot. Après que la fonction a traité un lot avec succès, Lambda supprime ses messages de la file d'attente.

Quand Lambda reçoit un lot, les messages restent dans la file d'attente mais sont masqués pendant toute la durée du délai de visibilité de la file d'attente. Si votre fonction traite tous les messages d'un lot avec succès, Lambda supprime les messages depuis la file d'attente. Par défaut, si votre fonction rencontre une erreur lors du traitement d'un lot, tous les messages de ce lot redeviennent visibles dans la file d'attente une fois le délai de visibilité expiré. Pour cette raison, le code de votre fonction peut traiter le même message plusieurs fois sans effets secondaires involontaires.

> **Warning**
>
> Les mappages des sources d'événements Lambda traitent chaque événement au moins une fois, et le traitement des enregistrements peut être dupliqué. Pour éviter les problèmes potentiels liés à des événements dupliqués, nous vous recommandons vivement de rendre votre code de fonction idempotent. Pour en savoir plus, consultez Comment rendre ma fonction Lambda idempotente dans le Knowledge Center. AWS

Pour empêcher Lambda de traiter un message plusieurs fois, vous pouvez soit configurer le mappage de votre source d'événements pour inclure les défaillances d'éléments de lot dans la réponse de votre fonction, soit utiliser l'DeleteMessageAPI pour supprimer les messages de la file d'attente au fur et à mesure que votre fonction Lambda les traite correctement.

Pour de plus amples informations sur les paramètres de configuration pris en charge par Lambda pour les mappages des sources d'événements SQS, consultez the section called "Création d'un mappage des sources d'événements SQS".

Utilisation du mode provisionné avec les mappages de sources d'événements Amazon SQS

Pour les charges de travail où vous devez optimiser le débit de votre mappage des sources d'événements, vous pouvez utiliser le mode provisionné. En mode provisionné, vous définissez des limites minimales et maximales pour le nombre de sondeurs d'événements alloués. Ces sondeurs d'événements alloués sont dédiés à votre mappage des sources d'événements et peuvent gérer les pics de messages inattendus grâce à une mise à l'échelle automatique réactive. Le mappage des sources d'événements Amazon SQS configuré en mode provisionné évolue 3 fois plus vite (jusqu'à 1 000 appels simultanés par minute) et prend en charge une simultanéité 16 fois supérieure (jusqu'à 20 000 appels simultanés) par rapport à la capacité de mappage des sources d'événements Amazon SQS par défaut. Nous vous recommandons d'utiliser le mode provisionné pour les charges de travail basées sur les événements Amazon SQS soumises à des exigences de performance strictes, telles que les sociétés de services financiers traitant les flux de données du marché, les plateformes de commerce électronique fournissant des recommandations personnalisées en temps réel et les sociétés de jeux gérant les interactions avec les joueurs en direct. L'utilisation du mode alloué génère des coûts supplémentaires. Pour connaître les tarifs détaillés, consultez la section AWS Lambda des prix.

Chaque sondeur d'événements en mode provisionné peut gérer jusqu'à 1 % MB/s du débit, jusqu'à 10 appels simultanés ou jusqu'à 10 appels d'API d'interrogation Amazon SQS par seconde. La plage de valeurs acceptées pour le nombre minimum de sondeurs d'événements (MinimumPollers) est comprise entre 2 et 200, avec une valeur par défaut de 2. La plage de valeurs acceptées pour le nombre maximum de sondeurs d'événements (MaximumPollers) est comprise entre 2 et 2 000, avec une valeur par défaut de 200. MaximumPollers doit être supérieur ou égal à MinimumPollers.

Déterminer les sondeurs d'événements requis

Pour estimer le nombre de sondeurs d'événements nécessaires pour garantir des performances de traitement des messages optimales lors de l'utilisation du mode provisionné pour SQS ESM, collectez les indicateurs suivants pour votre application : pic d'événements SQS par seconde nécessitant un traitement à faible latence, taille moyenne de la charge utile des événements SQS, durée moyenne de la fonction Lambda et taille de lot configurée.

Vous pouvez d'abord estimer le nombre d'événements SQS par seconde (EPS) pris en charge par un sondeur d'événements pour votre charge de travail à l'aide de la formule suivante :

```
EPS per event poller =
```

```
        minimum(
            ceiling(1024 / average event size in KB),
            ceiling(10 / average function duration in seconds) * batch size,
            min(100, 10 * batch size)
                )
```

Ensuite, vous pouvez calculer le nombre minimum de sondeurs requis à l'aide de la formule ci-dessous. Ce calcul garantit que vous fournissez une capacité suffisante pour répondre à vos besoins de trafic de pointe.

```
Required event pollers = (Peak number of events per second in Queue) / EPS per event
 poller
```

Prenons l'exemple d'une charge de travail avec une taille de lot par défaut de 10, une taille moyenne des événements de 3 Ko, une durée de fonctionnement moyenne de 100 ms et une exigence de gestion de 1 000 événements par seconde. Dans ce scénario, chaque sondeur d'événements prendra en charge environ 100 événements par seconde (EPS). Par conséquent, vous devez définir un nombre minimum de sondeurs à 10 pour répondre de manière adéquate à vos besoins de pointe en matière de trafic. Si votre charge de travail présente les mêmes caractéristiques mais avec une durée de fonctionnement moyenne d'une seconde, chaque sondeur ne prendra en charge que 10 EPS, ce qui vous obligera à configurer au moins 100 sondeurs pour prendre en charge 1 000 événements par seconde avec une faible latence.

Nous recommandons d'utiliser une taille de lot par défaut de 10 ou plus pour optimiser l'efficacité des sondeurs d'événements en mode provisionné. Des lots de plus grande taille permettent à chaque sondeur de traiter un plus grand nombre d'événements par appel, ce qui améliore le débit et la rentabilité. Lorsque vous planifiez votre capacité de sondeurs d'événements, tenez compte des pics de trafic potentiels et envisagez de définir une valeur de minimumPollers légèrement supérieure au minimum calculé pour créer une zone tampon. En outre, surveillez les caractéristiques de votre charge de travail au fil du temps, car les modifications de la taille des messages, de la durée des fonctions ou des modèles de trafic peuvent nécessiter des ajustements de la configuration de votre sondeur d'événements afin de maintenir des performances et une rentabilité optimales. Pour une planification précise de la capacité, nous vous recommandons de tester votre charge de travail spécifique afin de déterminer l'EPS réel que chaque enquêteur peut piloter.

Configuration du mode provisionné pour le mappage des sources d'événements Amazon SQS

Vous pouvez configurer le mode provisionné pour le mappage de vos sources d'événements Amazon SQS à l'aide de la console ou de l'API Lambda.

Pour configurer le mode provisionné pour un mappage de source d'événements Amazon SQS existant (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez la fonction avec le mappage des sources d'événements Amazon SQS pour laquelle vous souhaitez configurer le mode provisionné.
3. Choisissez Configuration, puis Déclencheurs.
4. Choisissez le mappage des sources d'événements Amazon SQS pour lequel vous souhaitez configurer le mode provisionné, puis choisissez Modifier.
5. Sous Configuration du mappage des sources d’événements, choisissez Configurer le mode provisionné.
   - Pour le nombre minimum de sondeurs d'événements, entrez une valeur comprise entre 2 et 200. Si vous ne spécifiez aucune valeur, Lambda choisit une valeur par défaut de 2.
   - Pour le nombre maximum de sondeurs d'événements, entrez une valeur comprise entre 2 et 2 000. Cette valeur doit être supérieure ou égale à la valeur du Nombre minimal de sondeurs d’événements. Si vous ne spécifiez aucune valeur, Lambda choisit la valeur par défaut 200.
6. Choisissez Enregistrer.

Vous pouvez configurer le mode provisionné par programmation à l'aide de l'`ProvisionedPollerConfig`objet de votre. `EventSourceMappingConfiguration` Par exemple, la commande `UpdateEventSourceMapping` CLI suivante configure une `MinimumPollers` valeur de 5 et une `MaximumPollers` valeur de 100.

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --provisioned-poller-config '{"MinimumPollers": 5, "MaximumPollers": 100}'
```

Après avoir configuré le mode alloué, vous pouvez observer l'utilisation des sondeurs d'événements pour votre charge de travail en surveillant la métrique `ProvisionedPollers`. Pour plus d'informations, consultez la section Métriques de mappage des sources d'événements.

Pour désactiver le mode provisionné et revenir au mode par défaut (à la demande), vous pouvez utiliser la commande `UpdateEventSourceMapping` CLI suivante :

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --provisioned-poller-config '{}'
```

> **Note**
>
> Le mode provisionné ne peut pas être utilisé conjointement avec le paramètre de simultanéité maximale. Lorsque vous utilisez le mode provisionné, vous contrôlez la simultanéité maximale grâce au nombre maximal de sondeurs d'événements.

Pour plus d'informations sur la configuration du mode provisionné, consultezthe section called "Création d'un mappage".

Exemple d'événement de message de file d'attente standard

Example Événement de message Amazon SQS (file d'attente standard)

```
{
    "Records": [
        {
            "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
            "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
            "body": "Test message.",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082649183",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082649185"
            },
            "messageAttributes": {
                "myAttribute": {
                    "stringValue": "myValue",
```

```
                    "stringListValues": [],
                    "binaryListValues": [],
                    "dataType": "String"
                }
            },
            "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
            "awsRegion": "us-east-2"
        },
        {
            "messageId": "2e1424d4-f796-459a-8184-9c92662be6da",
            "receiptHandle": "AQEBzWwaftRI0KuVm4tP+/7q1rGgNqicHq...",
            "body": "Test message.",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082650636",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082650649"
            },
            "messageAttributes": {},
            "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
            "awsRegion": "us-east-2"
        }
    ]
}
```

Par défaut, Lambda interroge jusqu'à 10 messages à la fois dans votre file d'attente, et envoie ce lot à votre fonction. Pour éviter d'invoquer la fonction avec un petit nombre d'enregistrements, vous pouvez configurer la source d'événement de manière à les mettre en mémoire tampon pendant 5 minutes en configurant une fenêtre de lot. Avant d'invoquer la fonction, Lambda continue d'interroger les messages de la file d'attente standard jusqu'à ce que la fenêtre de lot expire, que le quota de taille des données utiles de l'invocation soit atteint ou que la taille maximale configurée du lot soit atteinte.

Si vous utilisez une fenêtre de lot et que votre file d'attente SQS contient un trafic très faible, Lambda peut attendre 20 secondes avant d'invoquer votre fonction. C'est le cas même si vous définissez une fenêtre de lot inférieure à 20 secondes.

> **Note**
>
> Sous Java, vous pouvez rencontrer des erreurs de pointeur nul lors de la désérialisation de JSON. Cela peut être dû à la façon dont les cas « Records » (Enregistrements) et « eventSourceARN » sont convertis par le mappeur d'objets JSON.

Exemple d'événement de message de file d'attente FIFO

Pour les files d'attente FIFO, les enregistrements contiennent des attributs supplémentaires liés à la déduplication et au séquençage.

Example Événement de message Amazon SQS (file d'attente FIFO)

```
{
    "Records": [
        {
            "messageId": "11d6ee51-4cc7-4302-9e22-7cd8afdaadf5",
            "receiptHandle": "AQEBBX8nesZEXmkhsmZeyIE8iQAMig7qw...",
            "body": "Test message.",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1573251510774",
                "SequenceNumber": "18849496460467696128",
                "MessageGroupId": "1",
                "SenderId": "AIDAIO23YVJENQZJOL4VO",
                "MessageDeduplicationId": "1",
                "ApproximateFirstReceiveTimestamp": "1573251510774"
            },
            "messageAttributes": {},
            "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:fifo.fifo",
            "awsRegion": "us-east-2"
        }
    ]
}
```

Création et configuration d'un mappage des sources d'événements Amazon SQS

Pour traiter les messages Amazon SQS avec Lambda, configurez votre file d'attente avec les paramètres appropriés, puis créez un mappage des sources d'événements Lambda.

Rubriques

- Configuration d'une file d'attente à utiliser avec Lambda
- Configuration des autorisations de rôle d'exécution Lambda
- Création d'un mappage des sources d'événements SQS

Configuration d'une file d'attente à utiliser avec Lambda

Si vous n'avez pas encore de file d'attente Amazon SQS, créez-en une à utiliser en tant que source d'événement pour votre fonction Lambda. La fonction Lambda et la file d'attente Amazon SQS doivent être Région AWS identiques, bien qu'elles puissent être différentes. Comptes AWS

Pour laisser à votre fonction le temps de traiter chaque lot d'enregistrements, définissez le délai de visibilité de la file d'attente source à au moins six fois le délai de configuration sur votre fonction. Le délai supplémentaire permet à Lambda d'effectuer une nouvelle tentative si l'exécution de la fonction est limitée pendant le traitement d'un lot précédent.

> **Note**
>
> Le délai d'expiration de votre fonction doit être inférieur ou égal au délai d'expiration de visibilité de la file d'attente. Lambda valide cette exigence lorsque vous créez ou mettez à jour un mappage des sources d'événements, et renvoie une erreur si le délai d'expiration de la fonction dépasse le délai d'expiration de visibilité de la file d'attente.

Par défaut, si Lambda rencontre une erreur à un moment donné lors du traitement d'un lot, tous les messages de ce lot retournent dans la file d'attente. Après le délai de visibilité, les messages redeviennent visibles pour Lambda. Vous pouvez configurer le mappage des sources d'événements pour utiliser les réponses par lots partielles afin que seuls les messages ayant échoué soient renvoyés dans la file d'attente. En outre, en cas d'échec répété du traitement d'un message, Amazon SQS peut envoyer celui-ci à une file d'attente de lettres mortes. Nous vous recommandons de définir `maxReceiveCount` sur la stratégie de redirection de votre file d'attente source sur au moins 5. Cela

donne à Lambda la possibilité d'effectuer quelques tentatives avant d'envoyer les messages ayant échoué directement à la file d'attente de lettres mortes.

Configuration des autorisations de rôle d'exécution Lambda

La politique AWSLambdaSQSQueueExecutionRole AWS gérée inclut les autorisations dont Lambda a besoin pour lire depuis votre file d'attente Amazon SQS. Vous pouvez ajouter cette politique gérée au rôle d'exécution de votre fonction.

En option, si vous utilisez une file d'attente chiffrée, vous devez également ajouter l'autorisation suivante à votre rôle d'exécution :

- kms:Decrypt

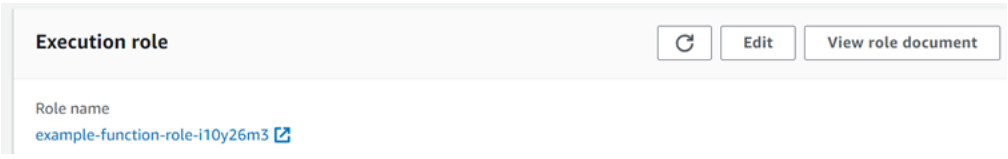
Création d'un mappage des sources d'événements SQS

Créez un mappage de source d'événement pour indiquer à Lambda d'envoyer des éléments de votre file d'attente à une fonction Lambda. Vous pouvez créer plusieurs mappages de source d'événement pour traiter des éléments de plusieurs files d'attente avec une seule fonction. Quand Lambda invoque la fonction cible, l'événement peut contenir plusieurs éléments, jusqu'à une taille de lot maximale configurable.

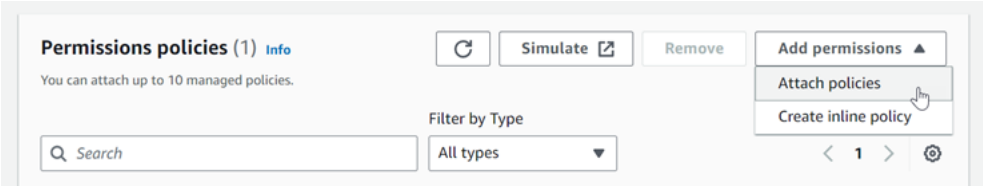
Pour configurer votre fonction afin qu'elle puisse lire depuis Amazon SQS, associez la politique AWSLambdaSQSQueueExecutionRole AWS gérée à votre rôle d'exécution. Créez ensuite un mappage des sources d'événements SQS à partir de la console en suivant les étapes suivantes.

Pour ajouter des autorisations et créer un déclencheur

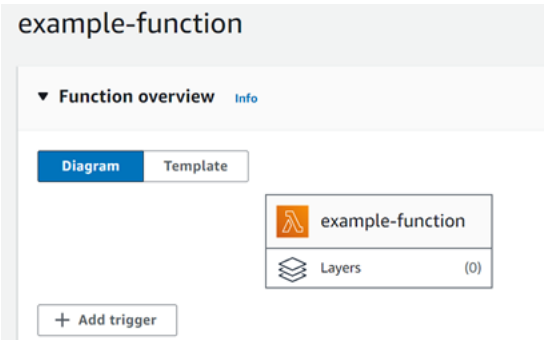
1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom d'une fonction.
3. Choisissez l'onglet Configuration, puis Permissions (Autorisations).
4. Sous Nom du rôle, cliquez sur le lien vers votre rôle d'exécution. Ce lien ouvre le rôle dans la console IAM.



5. Choisissez Ajouter des autorisations, puis Attacher des politiques.



6. Dans le champ de recherche, entrez `AWSLambdaSQSQueueExecutionRole`. Ajoutez cette politique à votre rôle d'exécution. Il s'agit d'une politique AWS gérée qui contient les autorisations dont votre fonction a besoin pour lire depuis une file d'attente Amazon SQS. Pour plus d'informations sur cette politique, consultez AWSLambdaSQSQueueExecutionRolela référence des politiques AWS gérées.
7. Revenez à votre fonction dans la console Lambda. Sous Function overview (Vue d'ensemble de la fonction), choisissez Add trigger (Ajouter un déclencheur).



8. Choisissez un type de déclencheur.
9. Configurez les options requises, puis choisissez Add (Ajouter).

Lambda prend en charge les options de configuration suivantes pour les sources d'événements Amazon SQS :

File d'attente SQS

La file d'attente Amazon SQS à partir de laquelle lire les enregistrements. La fonction Lambda et la file d'attente Amazon SQS doivent être Région AWS identiques, bien qu'elles puissent être différentes. Comptes AWS

Activation du déclencheur

L'état du mappage des sources d'événements. Activez le déclencheur est sélectionné par défaut.

Taille de lot

Le nombre maximum d'enregistrements à envoyer à la fonction dans chaque lot. Pour une file d'attente standard, cela peut aller jusqu'à 10 000 registres. Pour une file d'attente FIFO,

le maximum est de 10. Pour une taille de lot supérieure à 10, vous devez également définir la fenêtre de lot (`MaximumBatchingWindowInSeconds`) sur au moins 1 seconde.

Configurez le délai d'attente de votre fonction de façon à laisser suffisamment de temps pour traiter le lot entier d'éléments. Si les éléments sont longs à traiter, choisissez une taille de lot plus petite. Une grande taille de lot peut améliorer l'efficacité pour des charges de travail qui sont très rapides ou qui induisent beaucoup d'efforts supplémentaires. Si vous configurez une simultanéité réservée sur votre fonction, définissez un minimum de cinq exécutions simultanées pour réduire le risque d'erreurs de limitation lorsque Lambda invoque votre fonction.

Lambda transmet tous les registres du lot à la fonction en un seul appel, tant que la taille totale des événements ne dépasse pas le quota de taille des données utiles d'invocation pour une invocation synchrone (6 Mo). Des métadonnées sont générées par Lambda et Amazon SQS pour chaque registre. Ces métadonnées supplémentaires sont comptabilisées dans la taille de charge utile totale, ce qui peut entraîner l'envoi dans un lot d'un nombre total d'enregistrements inférieur à la taille du lot configuré. Les champs de métadonnées qu'Amazon SQS envoie peuvent être de longueur variable. Pour plus d'informations sur les champs de métadonnées Amazon SQS, consultez la documentation relative aux opérations d'ReceiveMessageAPI dans le manuel Amazon Simple Queue Service API Reference.

Fenêtre de lot

Intervalle de temps maximal (en secondes) pour collecter des enregistrements avant d'invoquer la fonction. Cela s'applique uniquement aux files d'attente standards.

Si vous utilisez une fenêtre de lot supérieure à 0 seconde, vous devez tenir compte de l'augmentation du temps de traitement dans le délai de visibilité de votre file d'attente. Nous vous recommandons de paramétrer votre délai de visibilité de file d'attente à six fois le délai d'expiration de la fonction, en plus de la valeur de `MaximumBatchingWindowInSeconds`. Cela permet à votre fonction Lambda de traiter chaque lot d'événements et de réessayer en cas d'erreur de limitation.

Lorsque les messages sont disponibles, Lambda commence à les traiter par lots. Lambda commence à traiter cinq lots à la fois avec cinq invocations simultanés de votre fonction. Si les messages sont toujours disponibles, Lambda ajoute jusqu'à 300 appels simultanés de votre fonction par minute, jusqu'à un maximum de 1 250 appels simultanés. En mode provisionné, chaque sondeur d'événements peut gérer jusqu'à 1 % MB/s du débit, jusqu'à 10 appels simultanés ou jusqu'à 10 appels d'API d'interrogation Amazon SQS par seconde. Lambda adapte le nombre de sondeurs d'événements entre le minimum et le maximum configurés, en

ajoutant rapidement jusqu'à 1 000 appels simultanés par minute afin de fournir un traitement à faible latence de vos événements Amazon SQS. Vous pouvez contrôler la mise à l'échelle et la simultanéité grâce à ces paramètres minimaux et maximaux d'un sondeur d'événements. Pour en savoir plus sur la mise à l'échelle et la simultanéité des fonctions, consultezPrésentation de la mise à l'échelle de fonction Lambda.

Pour traiter plus de messages, vous pouvez optimiser votre fonction Lambda pour un débit plus élevé. Pour plus d'informations, consultez Comprendre comment s' AWS Lambda adapte aux files d'attente standard Amazon SQS.

Critères de filtrage

Ajoutez des critères de filtrage pour contrôler les événements que Lambda envoie à votre fonction pour traitement. Pour de plus amples informations, veuillez consulter Contrôle des événements envoyés par Lambda à votre fonction.

Simultanéité maximum

Le nombre maximum de fonctions simultanées que la source d'événement peut invoquer. Ne peut pas être utilisé lorsque le mode provisionné est activé. Pour de plus amples informations, veuillez consulter Configuration de la simultanéité maximale pour les sources d'événements Amazon SQS.

Mode alloué

Lorsque cette option est activée, alloue des ressources de sondage dédiées au mappage des sources de votre événement. Vous pouvez configurer le nombre minimum (2-200) et maximum (2-2000) de sondeurs d'événements. Chaque sondeur d'événements peut gérer jusqu'à 1 % MB/sec du débit, jusqu'à 10 appels simultanés ou jusqu'à 10 appels d'API d'interrogation Amazon SQS par seconde.

> **Note**
>
> Remarque : vous ne pouvez pas utiliser simultanément le mode provisionné et la simultanéité maximale. Lorsque le mode provisionné est activé, utilisez le paramètre du nombre maximum de sondeurs pour contrôler la simultanéité.

Configuration du comportement de mise à l’échelle pour les mappages des sources d’événements SQS

Vous pouvez contrôler le comportement de dimensionnement de vos mappages de sources d'événements Amazon SQS via des paramètres de simultanéité maximale ou en activant le mode provisionné. Ces options s'excluent mutuellement.

Par défaut, Lambda adapte automatiquement les sondeurs d'événements en fonction du volume des messages. Lorsque vous activez le mode provisionné, vous allouez un nombre minimum et maximum de ressources de sondage dédiées qui restent prêtes à gérer les modèles de trafic attendus. Cela vous permet d'optimiser les performances de votre mappage des sources d'événements de deux manières :

- Mode standard (par défaut) : Lambda gère automatiquement le dimensionnement, en commençant par un petit nombre de sondeurs, puis en augmentant ou en diminuant en fonction de la charge de travail.
- Mode provisionné : vous configurez des ressources d'interrogation dédiées avec des limites minimales et maximales, ce qui permet une mise à l'échelle 3 fois plus rapide et une capacité de traitement jusqu'à 16 fois supérieure.

Pour les files d’attente standard, Lambda utilise une interrogation longue pour interroger une file d’attente jusqu’à ce qu’elle devienne active. Lorsque les messages sont disponibles, Lambda commence à traiter cinq lots à la fois avec cinq invocations simultanées de votre fonction. Si les messages sont toujours disponibles, Lambda augmente le nombre de processus lisant des lots de 300 appels simultanés supplémentaires par minute. Le nombre maximum d'appels qu'un mappage de source d'événement peut traiter simultanément est de 1 250. Lorsque le trafic est faible, Lambda réduit le traitement à cinq appels simultanés et peut l'optimiser jusqu'à deux appels simultanés afin de réduire les appels Amazon SQS et les coûts correspondants. Toutefois, cette optimisation n’est pas disponible lorsque vous activez le paramètre de simultanéité maximale.

Pour les files d’attente FIFO, Lambda envoie les messages à votre fonction dans l’ordre de leur réception. Lorsque vous envoyez un message à une file d’attente FIFO, vous spécifiez un ID de groupe de messages. Amazon SQS veille à ce que les messages du même groupe soient livrés à Lambda dans l’ordre. Lorsque Lambda lit vos messages par lots, chaque lot peut contenir des messages provenant de plusieurs groupes de messages, mais l’ordre des messages est conservé. Si la fonction renvoie une erreur, toutes les nouvelles tentatives sont effectuées sur les messages concernés avant que Lambda reçoive des messages supplémentaires du même groupe.

En mode provisionné, chaque sondeur d'événements peut gérer jusqu'à 1 % MB/sec du débit, jusqu'à 10 appels simultanés ou jusqu'à 10 appels d'API d'interrogation Amazon SQS par seconde. Lambda adapte le nombre d'interrogateurs d'événements entre le minimum et le maximum configurés, en ajoutant rapidement jusqu'à 1 000 appels simultanés par minute afin de fournir un traitement cohérent et à faible latence de vos événements Amazon SQS. L'utilisation du mode alloué génère des coûts supplémentaires. Pour connaître les tarifs détaillés, consultez AWS Lambda les tarifs. Chaque sondeur d'événements utilise de longs sondages dans votre file d'attente SQS, avec jusqu'à 10 sondages par seconde, ce qui entraîne des coûts liés aux demandes d'API SQS. Consultez la tarification d'Amazon SQS pour plus de détails. Vous pouvez contrôler la mise à l'échelle et la simultanéité par le biais de ces paramètres minimaux et maximaux d'un sondeur d'événements, plutôt que d'utiliser le paramètre de simultanéité maximale, car ces options ne peuvent pas être utilisées conjointement.

> **Note**
>
> Vous ne pouvez pas utiliser le paramètre de simultanéité maximale et le mode provisionné en même temps. Lorsque le mode provisionné est activé, vous contrôlez le dimensionnement et la simultanéité du mappage de votre source d'événements Amazon SQS via le nombre minimum et maximum de sondeurs d'événements.

Configuration de la simultanéité maximale pour les sources d'événements Amazon SQS

Vous pouvez utiliser le paramètre de simultanéité maximale pour contrôler le comportement de mise à l'échelle de vos sources d'événements SQS. Notez que la simultanéité maximale ne peut pas être utilisée lorsque le mode provisionné est activé. Le paramètre de simultanéité maximale limite le nombre d'instances simultanées de la fonction qu'une source d'événements Amazon SQS peut invoquer. La simultanéité maximale est un paramètre de niveau source d'événement. Si vous avez plusieurs sources d'événements Amazon SQS mappées à une fonction, chaque source d'événements peut avoir un paramètre de simultanéité maximale séparé. Vous pouvez utiliser la simultanéité maximale pour empêcher une file d'attente d'utiliser toute la simultanéité réservée de la fonction ou le reste du quota de simultanéité du compte. La configuration de la simultanéité maximale sur une source d'événements Amazon SQS est gratuite.

Il est important de noter que la simultanéité maximale et la simultanéité réservée sont deux paramètres indépendants. Ne définissez pas une simultanéité maximale supérieure à la simultanéité réservée de la fonction. Si vous configurez la simultanéité maximale, assurez-vous que la

simultanéité réservée de votre fonction est supérieure ou égale à la simultanéité maximale totale pour toutes les sources d'événements Amazon SQS sur la fonction. Sinon, Lambda peut limiter vos messages.

Lorsque le quota de simultanéité de votre compte est défini sur la valeur par défaut de 1 000, un mappage des sources d'événements Amazon SQS peut être mis à l'échelle pour invoquer des instances de fonction jusqu'à cette valeur, sauf si vous spécifiez une simultanéité maximale.

Si vous recevez une augmentation du quota de simultanéité par défaut de votre compte, Lambda risque de ne pas être en mesure d'invoquer des instances de fonctions concurrentes jusqu'à votre nouveau quota. Par défaut, Lambda peut effectuer une mise à l'échelle pour invoquer jusqu'à 1 250 instances de fonctions simultanées pour un mappage des sources d'événements Amazon SQS. Si cela ne correspond pas à votre cas d'utilisation, contactez le AWS support pour discuter d'une augmentation de la simultanéité du mappage des sources d'événements Amazon SQS de votre compte.

> **Note**
>
> Pour les files d'attente FIFO, les appels simultanés sont plafonnés soit par le nombre de groupes de messages IDs (`messageGroupId`), soit par le paramètre de simultanéité maximal, selon le plus faible des deux. Par exemple, si vous avez six groupes de messages IDs et que la simultanéité maximale est fixée à 10, votre fonction peut avoir un maximum de six appels simultanés.

Vous pouvez configurer la simultanéité maximale sur les mappages des sources d'événements Amazon SQS nouveaux et existants.

Configuration de la simultanéité maximale à l'aide de la console Lambda

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom d'une fonction.
3. Sous Function overview (Vue d'ensemble des fonctions), choisissez SQS. Cela ouvre l'onglet Configuration.
4. Sélectionnez le déclencheur Amazon SQS et choisissez Edit (Modifier).
5. Pour Maximum concurrency (Simultanéité maximale), saisissez un nombre compris entre 2 et 1 000. Pour désactiver la simultanéité maximale, laissez la case vide.
6. Choisissez Enregistrer.

Configurez la simultanéité maximale à l'aide de AWS Command Line Interface ()AWS CLI

Utilisez la commande update-event-source-mapping avec l'option `--scaling-config`. Exemple :

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --scaling-config '{"MaximumConcurrency":5}'
```

Pour désactiver la simultanéité maximale, entrez une valeur vide pour `--scaling-config` :

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --scaling-config "{}"
```

Configuration de la simultanéité maximale à l'aide de l'API Lambda

Utilisez l'UpdateEventSourceMappingaction CreateEventSourceMappingou avec un ScalingConfigobjet.

Gestion des erreurs pour une source d'événements SQS dans Lambda

Pour gérer les erreurs liées à une source d'événement SQS, Lambda utilise automatiquement une stratégie de nouvelle tentative associée à une stratégie de backoff. Vous pouvez également personnaliser le comportement de gestion des erreurs en configurant votre mappage des sources d'événements SQS de manière à renvoyer des réponses par lots partielles.

Stratégie d'interruption pour les invocations ayant échoué

Lorsqu'une invocation échoue, Lambda tente de réessayer l'invocation tout en mettant en œuvre une stratégie d'interruption. La stratégie d'interruption diffère légèrement selon que Lambda a rencontré l'échec à cause d'une erreur dans votre code de fonction, ou à cause de la limitation.

- Si votre code de fonction est à l'origine de l'erreur, Lambda arrête le traitement et effectue une nouvelle tentative d'invocation. Dans le même temps, Lambda réduit progressivement les tentatives en diminuant la quantité de simultanéité allouée à votre mappage des sources d'événements Amazon SQS. Une fois le délai de visibilité de votre file d'attente expiré, le message réapparaît dans la file d'attente.
- Si l'invocation échoue en raison d'une limitation, Lambda réduit progressivement les tentatives en diminuant la quantité de simultanéité allouée à votre mappage des sources d'événements Amazon

SQS. Lambda continue à réessayer le message jusqu'à ce que l'horodatage du message dépasse le délai de visibilité de votre file d'attente, auquel cas Lambda abandonne le message.

Mise en œuvre de réponses partielles de lot

Lorsque votre fonction Lambda rencontre une erreur lors du traitement d'un lot, tous les messages de ce lot redeviennent par défaut visibles dans la file d'attente, y compris les messages traités avec succès par Lambda. Par conséquent, votre fonction peut finir par traiter le même message plusieurs fois.

Pour éviter de retraiter les messages traités avec succès d'un lot en échec, vous pouvez configurer le mappage des sources d'événements pour que seuls les messages ayant échoué soient à nouveau visibles. C'est ce que l'on appelle une réponse partielle de lot. Pour activer les réponses partielles par lots, spécifiez `ReportBatchItemFailures` l'FunctionResponseTypesaction lors de la configuration du mappage des sources d'événements. Cela permet à votre fonction de renvoyer un succès partiel, ce qui peut contribuer à réduire le nombre de nouvelles tentatives inutiles sur les enregistrements.

> **Note**
>
> L'utilitaire Batch Processor de Powertools for AWS Lambda gère automatiquement toute la logique de réponse partielle par lots. Cet utilitaire simplifie la mise en œuvre de modèles de traitement par lots et réduit le code personnalisé nécessaire pour gérer correctement les échecs d'éléments de lots. Il est disponible pour Python, Java, TypeScript et .NET.

Lorsque `ReportBatchItemFailures` est activé, Lambda ne réduit pas la taille de l'interrogation des messages lorsque les invocations de fonctions échouent. Si vous vous attendez à ce que certains messages échouent et que vous ne voulez pas que ces échecs aient un impact sur le taux de traitement des messages, utilisez `ReportBatchItemFailures`.

> **Note**
>
> Gardez les points suivants à l'esprit lorsque vous utilisez des réponses partielles de lot :
>
> - Si la fonction génère une exception, l'ensemble du lot est considéré comme un échec complet.
> - Si vous utilisez cette fonctionnalité avec une file d'attente FIFO, votre fonction doit arrêter le traitement des messages après le premier échec et renvoyer tous les messages échoués

et non traités dans `batchItemFailures`. Cela permet de préserver l'ordre des messages dans votre file d'attente.

Pour activer les rapports partiels de lot

1. Consultez les bonnes pratiques pour la mise en œuvre des réponses partielles de lot.
2. Exécutez la commande suivante pour activer `ReportBatchItemFailures` pour votre fonction. Pour récupérer l'UUID du mappage de votre source d'événements, exécutez la list-event-source-mappings AWS CLI commande.

```
aws lambda update-event-source-mapping \
--uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
--function-response-types "ReportBatchItemFailures"
```

3. Mettez à jour le code de votre fonction afin de capturer toutes les exceptions et de renvoyer les messages d'échec dans une réponse JSON `batchItemFailures`. La `batchItemFailures` réponse doit inclure une liste de messages IDs, sous forme de valeurs `itemIdentifier` JSON.

   Supposons, par exemple, que vous disposiez d'un lot de cinq messages IDs `id1`, avec le message `id2id3`,`id4`,, et`id5`. Votre fonction traite avec succès `id1`, `id3` et `id5`. Pour rendre les messages `id2` et `id4` visibles de nouveau dans la file d'attente, votre fonction doit renvoyer la réponse suivante :

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "id2"
        },
        {
            "itemIdentifier": "id4"
        }
    ]
}
```

   Voici quelques exemples de code de fonction qui renvoie la liste des messages ayant échoué IDs dans le lot :

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots SQS avec Lambda à l'aide de .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;

// Assembly attribute to enable the Lambda function's JSON input to be
 converted into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJson
namespace sqsSample;

public class Function
{
    public async Task<SQSBatchResponse> FunctionHandler(SQSEvent evnt,
 ILambdaContext context)
    {
        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 List<SQSBatchResponse.BatchItemFailure>();
        foreach(var message in evnt.Records)
        {
            try
            {
                //process your message
                await ProcessMessageAsync(message, context);
            }
            catch (System.Exception)
            {
                //Add failed message identifier to the batchItemFailures list
```

```
                batchItemFailures.Add(new
 SQSBatchResponse.BatchItemFailure{ItemIdentifier=message.MessageId});
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        if (String.IsNullOrEmpty(message.Body))
        {
            throw new Exception("No Body in SQS Message.");
        }
        context.Logger.LogInformation($"Processed message {message.Body}");
        // TODO: Do interesting work based on the new message
        await Task.CompletedTask;
    }
}
```

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots SQS avec Lambda à l'aide de Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
```

```
)

func handler(ctx context.Context, sqsEvent events.SQSEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, message := range sqsEvent.Records {
  if len(message.Body) > 0 {
   // Your message processing condition here
   fmt.Printf("Successfully processed message: %s\n", message.Body)
  } else {
   // Message processing failed
   fmt.Printf("Failed to process message %s\n", message.MessageId)
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": message.MessageId})
  }
 }

 sqsBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
 }
 return sqsBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots SQS avec Lambda à l'aide de Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSBatchResponse;

import java.util.ArrayList;
import java.util.List;

public class ProcessSQSMessageBatch implements RequestHandler<SQSEvent,
 SQSBatchResponse> {
    @Override
    public SQSBatchResponse handleRequest(SQSEvent sqsEvent, Context context)
 {
        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<SQSBatchResponse.BatchItemFailure>();

        for (SQSEvent.SQSMessage message : sqsEvent.getRecords()) {
            try {
                //process your message
            } catch (Exception e) {
                //Add failed message identifier to the batchItemFailures
 list
                batchItemFailures.add(new
 SQSBatchResponse.BatchItemFailure(message.getMessageId()));
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }
}
```

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signaler les défaillances d'éléments de lot SQS avec JavaScript Lambda à l'aide de.

```
// Node.js 20.x Lambda runtime, AWS SDK for Javascript V3
export const handler = async (event, context) => {
    const batchItemFailures = [];
    for (const record of event.Records) {
        try {
            await processMessageAsync(record, context);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }
    return { batchItemFailures };
};

async function processMessageAsync(record, context) {
    if (record.body && record.body.includes("error")) {
        throw new Error("There is an error in the SQS Message.");
    }
    console.log(`Processed message: ${record.body}`);
}
```

Signaler les défaillances d'éléments de lot SQS avec TypeScript Lambda à l'aide de.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, SQSBatchResponse, Context, SQSBatchItemFailure,
 SQSRecord } from 'aws-lambda';

export const handler = async (event: SQSEvent, context: Context):
 Promise<SQSBatchResponse> => {
    const batchItemFailures: SQSBatchItemFailure[] = [];

    for (const record of event.Records) {
        try {
            await processMessageAsync(record);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }
```

```
    return {batchItemFailures: batchItemFailures};
};

async function processMessageAsync(record: SQSRecord): Promise<void> {
    if (record.body && record.body.includes("error")) {
        throw new Error('There is an error in the SQS Message.');
    }
    console.log(`Processed message ${record.body}`);
}
```

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots SQS avec Lambda à l'aide de PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

use Bref\Context\Context;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }
```

```
    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        $this->logger->info("Processing SQS records");
        $records = $event->getRecords();

        foreach ($records as $record) {
            try {
                // Assuming the SQS message is in JSON format
                $message = json_decode($record->getBody(), true);
                $this->logger->info(json_encode($message));
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $this->markAsFailed($record);
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords SQS
 records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots SQS avec Lambda à l'aide de Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

def lambda_handler(event, context):
    if event:
        batch_item_failures = []
        sqs_batch_response = {}

        for record in event["Records"]:
            try:
                print(f"Processed message: {record['body']}")
            except Exception as e:
                batch_item_failures.append({"itemIdentifier":
 record['messageId']})

        sqs_batch_response["batchItemFailures"] = batch_item_failures
        return sqs_batch_response
```

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots SQS avec Lambda à l'aide de Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'json'

def lambda_handler(event:, context:)
  if event
    batch_item_failures = []
```

```
    sqs_batch_response = {}

    event["Records"].each do |record|
      begin
        # process message
      rescue StandardError => e
        batch_item_failures << {"itemIdentifier" => record['messageId']}
      end
    end

    sqs_batch_response["batchItemFailures"] = batch_item_failures
    return sqs_batch_response
  end
end
```

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots SQS avec Lambda à l'aide de Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::sqs::{SqsBatchResponse, SqsEvent},
    sqs::{BatchItemFailure, SqsMessage},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn process_record(_: &SqsMessage) -> Result<(), Error> {
    Err(Error::from("Error processing message"))
}
```

```
async fn function_handler(event: LambdaEvent<SqsEvent>) ->
 Result<SqsBatchResponse, Error> {
    let mut batch_item_failures = Vec::new();
    for record in event.payload.records {
        match process_record(&record).await {
            Ok(_) => (),
            Err(_) => batch_item_failures.push(BatchItemFailure {
                item_identifier: record.message_id.unwrap(),
            }),
        }
    }

    Ok(SqsBatchResponse {
        batch_item_failures,
    })
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(function_handler)).await
}
```

Si les événements ayant échoué ne retournent pas dans la file d'attente, voir Comment résoudre les problèmes liés à la fonction Lambda SQS ? ReportBatchItemFailures dans le AWS Knowledge Center.

Conditions de réussite et d’échec

Lambda traite un lot comme un succès complet si votre fonction renvoie l’un des éléments suivants :

- Une liste `batchItemFailures` vide
- Une liste `batchItemFailures` nulle
- Une `EventResponse` vide
- Une `EventResponse` nulle

Lambda traite un lot comme un échec complet si votre fonction renvoie l’un des éléments suivants :

- Une réponse JSON non valide
- Une chaîne `itemIdentifier` vide

- Un `itemIdentifier` nul
- Un `itemIdentifier` avec un nom de clé incorrect
- Une valeur `itemIdentifier` avec un ID de message inexistant

CloudWatch métriques

Pour déterminer si votre fonction signale correctement les défaillances d'articles par lots, vous pouvez surveiller les métriques Amazon SQS `NumberOfMessagesDeleted` et `ApproximateAgeOfOldestMessage` Amazon SQS sur Amazon. CloudWatch

- `NumberOfMessagesDeleted` suit le nombre de messages supprimés de votre file d'attente. Si cela tombe à 0, cela indique que la réponse de votre fonction ne renvoie pas correctement les messages d'échec.
- `ApproximateAgeOfOldestMessage` suit combien de temps le message le plus ancien est resté dans votre file d'attente. Une forte augmentation de cette métrique peut indiquer que votre fonction ne renvoie pas correctement les messages d'échec.

Utilisation de Powertools pour le AWS Lambda traitement par lots

L'utilitaire de traitement par lots de Powertools for gère AWS Lambda automatiquement la logique de réponse partielle par lots, réduisant ainsi la complexité de la mise en œuvre du signalement des défaillances par lots. Voici des exemples d'utilisation de l'utilitaire de traitement par lots :

Python

> **Note**
>
> Pour des exemples complets et des instructions de configuration, consultez la documentation de l'utilitaire de traitement par lots.

Traitement des messages Amazon SQS à l'aide d'un AWS Lambda traitement par lots.

```
import json
from aws_lambda_powertools import Logger
from aws_lambda_powertools.utilities.batch import BatchProcessor, EventType,
 process_partial_response
from aws_lambda_powertools.utilities.data_classes import SQSEvent
```

```
from aws_lambda_powertools.utilities.typing import LambdaContext

processor = BatchProcessor(event_type=EventType.SQS)
logger = Logger()

def record_handler(record):
    logger.info(record)
    # Your business logic here
    # Raise an exception to mark this record as failed

def lambda_handler(event, context: LambdaContext):
    return process_partial_response(
        event=event,
        record_handler=record_handler,
        processor=processor,
        context=context
    )
```

TypeScript

> **Note**
>
> Pour des exemples complets et des instructions de configuration, consultez la documentation de l'utilitaire de traitement par lots.

Traitement des messages Amazon SQS à l'aide d'un AWS Lambda traitement par lots.

```
import { BatchProcessor, EventType, processPartialResponse } from '@aws-lambda-
powertools/batch';
import { Logger } from '@aws-lambda-powertools/logger';
import type { SQSEvent, Context } from 'aws-lambda';

const processor = new BatchProcessor(EventType.SQS);
const logger = new Logger();

const recordHandler = async (record: any): Promise<void> => {
    logger.info('Processing record', { record });
    // Your business logic here
    // Throw an error to mark this record as failed
};

export const handler = async (event: SQSEvent, context: Context) => {
```

```
    return processPartialResponse(event, recordHandler, processor, {
        context,
    });
};
```

Paramètres Lambda pour les mappages des sources d’événement Amazon SQS

Tous les types de sources d'événements Lambda partagent les mêmes opérations CreateEventSourceMappinget les mêmes opérations d'UpdateEventSourceMappingAPI. Cependant, seuls certains paramètres s’appliquent à Amazon SQS.

Paramètre	Obligatoire	Par défaut	Remarques
BatchSize	N	10	Pour les files d’attente standard, le maximum est de 10 000. Pour les files d’attente FIFO, le maximum est de 10.
Activé	N	VRAI	Aucune
EventSourceArn	Y	N/A	ARN du flux de données ou d’un consommateur de flux
FunctionName	Y	N/A	Aucune
FilterCriteria	N	N/A	Contrôle des événements envoyés par Lambda à votre fonction
FunctionResponseTypes	N	N/A	Pour permettre à votre fonction de signaler des échecs spécifiques dans

Paramètre	Obligatoire	Par défaut	Remarques
			un lot, incluez la valeur `ReportBatchItemFailures` dans `FunctionResponseTypes` . Pour de plus amples informations, veuillez consulter Mise en œuvre de réponses partielles de lot.
MaximumBatchingWindowInSeconds	N	0	La fenêtre de traitement par lots n'est pas prise en charge pour les files d'attente FIFO
ProvisionedPollerConfig	N	N/A	Configure le nombre minimum (2-200) et maximum (2-2000) de sondeurs d'événements dédiés pour le mappage des sources d'événements SQS. Chaque sondeur peut gérer jusqu'à 1 % MB/sec du débit et 10 appels simultanés.
ScalingConfig	N	N/A	Configuration de la simultanéité maximale pour les sources d'événements Amazon SQS

Utilisation du filtrage des événements avec une source d'événement Amazon SQS

Vous pouvez utiliser le filtrage d'événements pour contrôler les enregistrements d'un flux ou d'une file d'attente que Lambda envoie à votre fonction. Pour obtenir des informations générales sur le fonctionnement du filtrage des événements, consultez the section called "Filtrage des événements".

Cette section porte sur le filtrage des événements pour les sources Amazon SQS.

> **Note**
>
> Les mappages des sources d'événements Amazon SQS prennent uniquement en charge le filtrage sur la clé `body`.

Rubriques

- Notions de base du filtrage des événements Amazon SQS

Notions de base du filtrage des événements Amazon SQS

Supposons que votre file d'attente Amazon SQS contienne des messages au format JSON suivant.

```
{
    "RecordNumber": 1234,
    "TimeStamp": "yyyy-mm-ddThh:mm:ss",
    "RequestCode": "AAAA"
}
```

Un exemple d'enregistrement pour cette file d'attente ressemblerait à ce qui suit.

```
{
    "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
    "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
    "body": "{\n "RecordNumber": 1234,\n "TimeStamp": "yyyy-mm-ddThh:mm:ss",\n
 "RequestCode": "AAAA"\n}",
    "attributes": {
        "ApproximateReceiveCount": "1",
        "SentTimestamp": "1545082649183",
```

```
        "SenderId": "AIDAIENQZJOLO23YVJ4VO",
        "ApproximateFirstReceiveTimestamp": "1545082649185"
        },
    "messageAttributes": {},
    "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
    "eventSource": "aws:sqs",
    "eventSourceARN": "arn:aws:sqs:us-west-2:123456789012:my-queue",
    "awsRegion": "us-west-2"
}
```

Pour filtrer en fonction du contenu de vos messages Amazon SQS, utilisez la clé `body` dans l'enregistrement du message Amazon SQS. Supposons que vous vouliez traiter uniquement les enregistrements où le `RequestCode` de votre message Amazon SQS est « BBBB ». L'objet `FilterCriteria` serait le suivant.

```
{
    "Filters": [
        {
            "Pattern": "{ \"body\" : { \"RequestCode\" : [ \"BBBB\" ] } }"
        }
    ]
}
```

Pour plus de clarté, voici la valeur du `Pattern` de filtre étendu en JSON simple :

```
{
    "body": {
        "RequestCode": [ "BBBB" ]
        }
}
```

Vous pouvez ajouter votre filtre à l'aide de la console, d'AWS CLI ou d'un modèle AWS SAM.

Console

Pour ajouter ce filtre à l'aide de la console, suivez les instructions de Attacher des critères de filtre à un mappage de sources d'événements (console) et saisissez la chaîne suivante comme critère de filtre.

```
{ "body" : { "RequestCode" : [ "BBBB" ] } }
```

AWS CLI

Pour créer un nouveau mappage des sources d'événements avec ces critères de filtrage à l'aide de l'outil AWS Command Line Interface (AWS CLI), exécutez la commande suivante.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RequestCode\" :
 [ \"BBBB\" ] } }"}]}'
```

Pour ajouter ces critères de filtre à un mappage des sources d'événements existant, exécutez la commande suivante.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RequestCode\" :
 [ \"BBBB\" ] } }"}]}'
```

AWS SAM

Pour ajouter ce filtre à l'aide d'AWS SAM, ajoutez l'extrait suivant au modèle YAML de votre source d'événement.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "body" : { "RequestCode" : [ "BBBB" ] } }'
```

Supposons que vous vouliez que votre fonction ne traite que les enregistrements où `RecordNumber` est supérieur à 9 999. L'objet `FilterCriteria` serait le suivant.

```
{
    "Filters": [
        {
            "Pattern": "{ \"body\" : { \"RecordNumber\" : [ { \"numeric\": [ \">\",
 9999 ] } ] } }"
        }
    ]
}
```

Pour plus de clarté, voici la valeur du `Pattern` de filtre étendu en JSON simple :

```
{
    "body": {
        "RecordNumber": [
            {
                "numeric": [ ">", 9999 ]
            }
        ]
    }
}
```

Vous pouvez ajouter votre filtre à l'aide de la console, d'AWS CLI ou d'un modèle AWS SAM.

Console

Pour ajouter ce filtre à l'aide de la console, suivez les instructions de Attacher des critères de filtre à un mappage de sources d'événements (console) et saisissez la chaîne suivante comme critère de filtre.

```
{ "body" : { "RecordNumber" : [ { "numeric": [ ">", 9999 ] } ] } }
```

AWS CLI

Pour créer un nouveau mappage des sources d'événements avec ces critères de filtrage à l'aide de l'outil AWS Command Line Interface (AWS CLI), exécutez la commande suivante.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RecordNumber\" :
 [ { \"numeric\": [ \">\", 9999 ] } ] } }"}]}'
```

Pour ajouter ces critères de filtre à un mappage des sources d'événements existant, exécutez la commande suivante.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RecordNumber\" :
 [ { \"numeric\": [ \">\", 9999 ] } ] } }"}]}'
```

AWS SAM

Pour ajouter ce filtre à l'aide d'AWS SAM, ajoutez l'extrait suivant au modèle YAML de votre source d'événement.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "body" : { "RecordNumber" : [ { "numeric": [ ">", 9999 ] } ] } }'
```

Pour Amazon SQS, le corps du message peut être n'importe quelle chaîne. Toutefois, cela peut être problématique si votre `FilterCriteria` s'attend à ce que `body` se présente dans un format JSON valide. Le scénario inverse est également vrai, si le corps du message entrant est au format JSON, mais que votre critère de filtre s'attend à ce que `body` soit une chaîne de texte brut, cela peut entraîner un comportement inattendu.

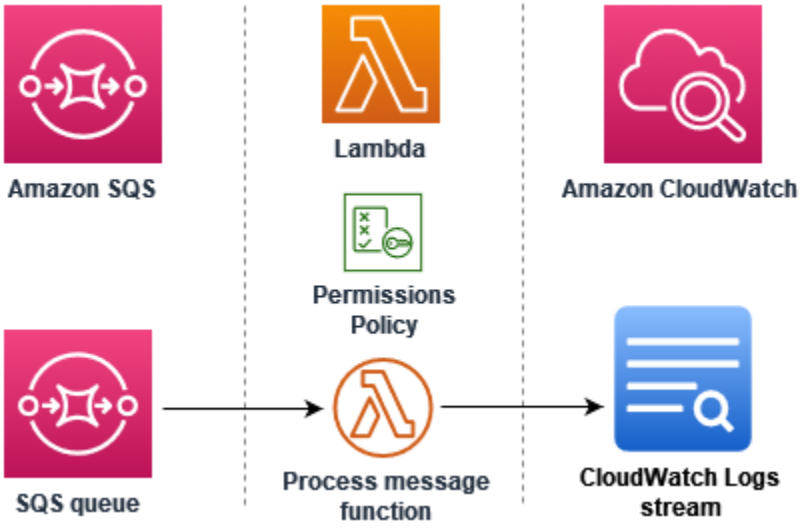
Pour éviter ce problème, assurez-vous que le format du corps dans vos `FilterCriteria` correspond au format attendu de `body` dans les messages que vous recevez de votre file d'attente. Avant de filtrer vos messages, Lambda évalue automatiquement le format du corps de votre message entrant et de votre modèle de filtre pour `body`. En cas de décalage, Lambda abandonne le message. Le tableau suivant résume cette évaluation :

Format du **body** du message entrant	Format du **body** du modèle de filtre	Action obtenue.
Chaîne de texte brut	Chaîne de texte brut	Lambda filtre en fonction de vos critères de filtre.
Chaîne de texte brut	Pas de modèle de filtre pour les propriétés des données	Lambda filtre (uniquement selon les autres propriétés de métadonnées) en fonction de vos critères de filtre.
Chaîne de texte brut	JSON valide	Lambda rejette le message.
JSON valide	Chaîne de texte brut	Lambda rejette le message.
JSON valide	Pas de modèle de filtre pour les propriétés des données	Lambda filtre (uniquement selon les autres propriétés de

Format du **body** du message entrant	Format du **body** du modèle de filtre	Action obtenue.
		métadonnées) en fonction de vos critères de filtre.
JSON valide	JSON valide	Lambda filtre en fonction de vos critères de filtre.

Didacticiel : utilisation de Lambda avec Amazon SQS

Dans ce didacticiel, vous créerez une fonction Lambda qui consomme les messages d'une file d'attente Amazon Simple Queue Service (Amazon SQS). La fonction Lambda s'exécute chaque fois qu'un nouveau message est ajouté à la file d'attente. La fonction écrit les messages dans un flux Amazon CloudWatch Logs. Le diagramme suivant montre les ressources AWS utilisées pour compléter ce tutoriel.



Pour compléter ce didacticiel, effectuez les tâches suivantes :

1. Créez une fonction Lambda qui écrit des messages dans Logs. CloudWatch
2. Créer une file d'attente Amazon SQS.
3. Créez un mappage des sources d'événements Lambda. Le mappage des sources d'événements lit la file d'attente Amazon SQS et invoque votre fonction Lambda lorsqu'un nouveau message est ajouté.

4. Testez la configuration en ajoutant des messages à votre file d'attente et en surveillant les résultats dans CloudWatch Logs.

Conditions préalables

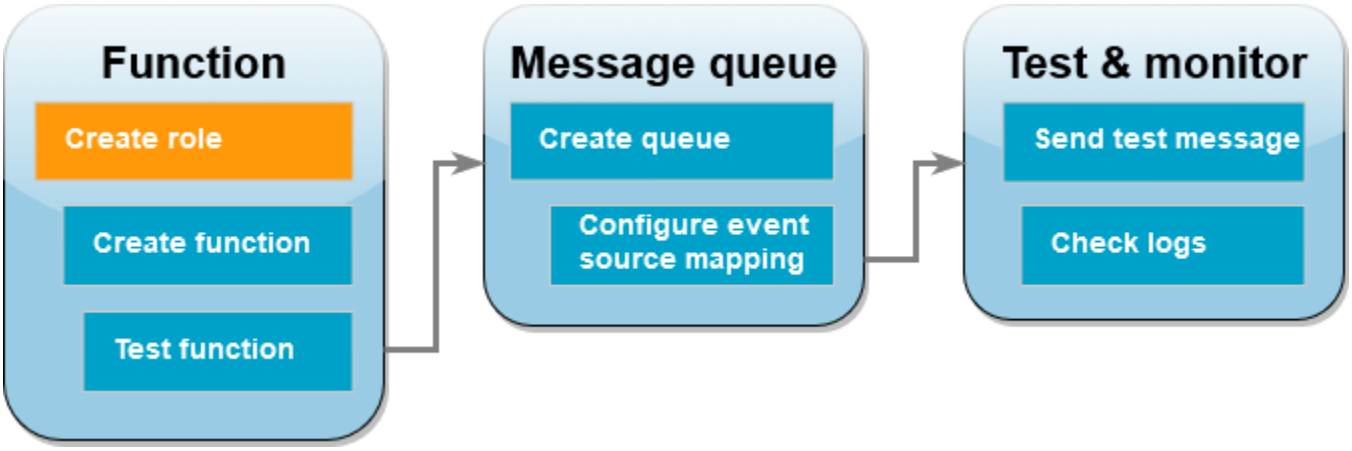
Installez le AWS Command Line Interface

Si vous ne l'avez pas encore installé AWS Command Line Interface, suivez les étapes décrites dans la section Installation ou mise à jour de la dernière version du AWS CLI pour l'installer.

Ce tutoriel nécessite un terminal de ligne de commande ou un shell pour exécuter les commandes. Sous Linux et macOS, utilisez votre gestionnaire de shell et de package préféré.

> **Note**
>
> Sous Windows, certaines commandes CLI Bash que vous utilisez couramment avec Lambda (par exemple `zip`) ne sont pas prises en charge par les terminaux intégrés du système d'exploitation. Installez le sous-système Windows pour Linux afin d'obtenir une version intégrée à Windows d'Ubuntu et Bash.

Créer le rôle d'exécution



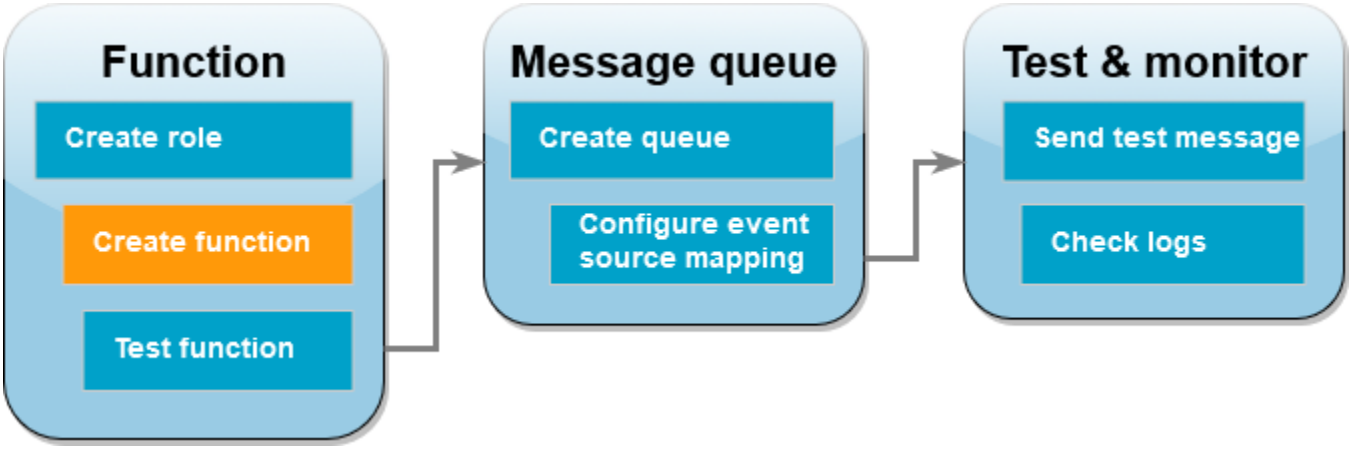
Un rôle d'exécution est un rôle Gestion des identités et des accès AWS (IAM) qui accorde à une fonction Lambda l'autorisation d' Services AWS accès et de ressources. Pour permettre à votre fonction de lire des éléments depuis Amazon SQS, joignez la politique d'AWSLambdaSQSQueueExecutionRoleautorisation.

Pour créer un rôle d'exécution et attacher une politique d'autorisations Amazon SQS

1. Ouvrez la page Rôles de la console IAM.
2. Sélectionnez Créer un rôle.
3. Pour Type d'entité de confiance, choisissez Service AWS .
4. Pour Cas d'utilisation, choisissez Lambda.
5. Choisissez Suivant.
6. Dans le champ de recherche Politiques d'autorisations, saisissez **`AWSLambdaSQSQueueExecutionRole`**.
7. Sélectionnez la AWSLambdaSQSQueueExecutionRolepolitique, puis cliquez sur Suivant.
8. Sous Détails du rôle, pour Nom du rôle, saisissez **`lambda-sqs-role`**, puis sélectionnez Créer un rôle.

Après la création du rôle, notez l'Amazon Resource Name (ARN) de votre rôle d'exécution. Vous en aurez besoin dans les étapes suivantes.

Créer la fonction



Créez une fonction Lambda qui traite vos messages Amazon SQS. Le code de fonction enregistre le corps du message Amazon SQS dans Logs. CloudWatch

Ce didacticiel utilise le runtime Node.js 24, mais nous avons également fourni des exemples de code dans d'autres langages d'exécution. Vous pouvez sélectionner l'onglet dans la zone suivante pour voir le code de l'exécution qui vous intéresse. Le JavaScript code que vous allez utiliser dans cette étape se trouve dans le premier exemple présenté dans l'JavaScriptonglet.

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement SQS avec Lambda en utilisant .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SqsIntegrationSampleCode
{
    public async Task FunctionHandler(SQSEvent evnt, ILambdaContext context)
    {
        foreach (var message in evnt.Records)
        {
            await ProcessMessageAsync(message, context);
        }

        context.Logger.LogInformation("done");
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed message {message.Body}");

            // TODO: Do interesting work based on the new message
```

```
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }

    }
}
```

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement SQS avec Lambda à l'aide de Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package integration_sqs_to_lambda

import (
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.SQSEvent) error {
 for _, record := range event.Records {
  err := processMessage(record)
  if err != nil {
   return err
  }
```

```
 }
 fmt.Println("done")
 return nil
}

func processMessage(record events.SQSMessage) error {
 fmt.Printf("Processed message %s\n", record.Body)
 // TODO: Do interesting work based on the new message
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement SQS avec Lambda à l'aide de Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSEvent.SQSMessage;

public class Function implements RequestHandler<SQSEvent, Void> {
    @Override
    public Void handleRequest(SQSEvent sqsEvent, Context context) {
        for (SQSMessage msg : sqsEvent.getRecords()) {
            processMessage(msg, context);
        }
        context.getLogger().log("done");
```

```
        return null;
    }

    private void processMessage(SQSMessage msg, Context context) {
        try {
            context.getLogger().log("Processed message " + msg.getBody());

            // TODO: Do interesting work based on the new message

        } catch (Exception e) {
            context.getLogger().log("An error occurred");
            throw e;
        }

    }
}
```

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement SQS avec JavaScript Lambda en utilisant.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message) {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
```

```
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

Consommation d'un événement SQS avec TypeScript Lambda en utilisant.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, Context, SQSHandler, SQSRecord } from "aws-lambda";

export const functionHandler: SQSHandler = async (
  event: SQSEvent,
  context: Context
): Promise<void> => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message: SQSRecord): Promise<any> {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement SQS avec Lambda à l'aide de PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\InvalidLambdaEvent;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $body = $record->getBody();
            // TODO: Do interesting work based on the new message
        }
```

```
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement SQS avec Lambda à l'aide de Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for message in event['Records']:
        process_message(message)
    print("done")

def process_message(message):
    try:
        print(f"Processed message {message['body']}")
        # TODO: Do interesting work based on the new message
    except Exception as err:
        print("An error occurred")
        raise err
```

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement SQS avec Lambda à l'aide de Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].each do |message|
    process_message(message)
  end
  puts "done"
end

def process_message(message)
  begin
    puts "Processed message #{message['body']}"
    # TODO: Do interesting work based on the new message
  rescue StandardError => err
    puts "An error occurred"
    raise err
  end
end
```

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement SQS avec Lambda à l'aide de Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sqs::SqsEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<SqsEvent>) -> Result<(), Error> {
    event.payload.records.iter().for_each(|record| {
        // process the record
        tracing::info!("Message body: {}",
 record.body.as_deref().unwrap_or_default())
    });

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

Pour créer une fonction Lambda Node.js.

1. Créez un répertoire pour le projet, puis passez à ce répertoire.

```
mkdir sqs-tutorial
cd sqs-tutorial
```

2. Copiez l'exemple de JavaScript code dans un nouveau fichier nomméindex.js.
3. Créez un package de déploiement à l'aide de la commande zip suivante.

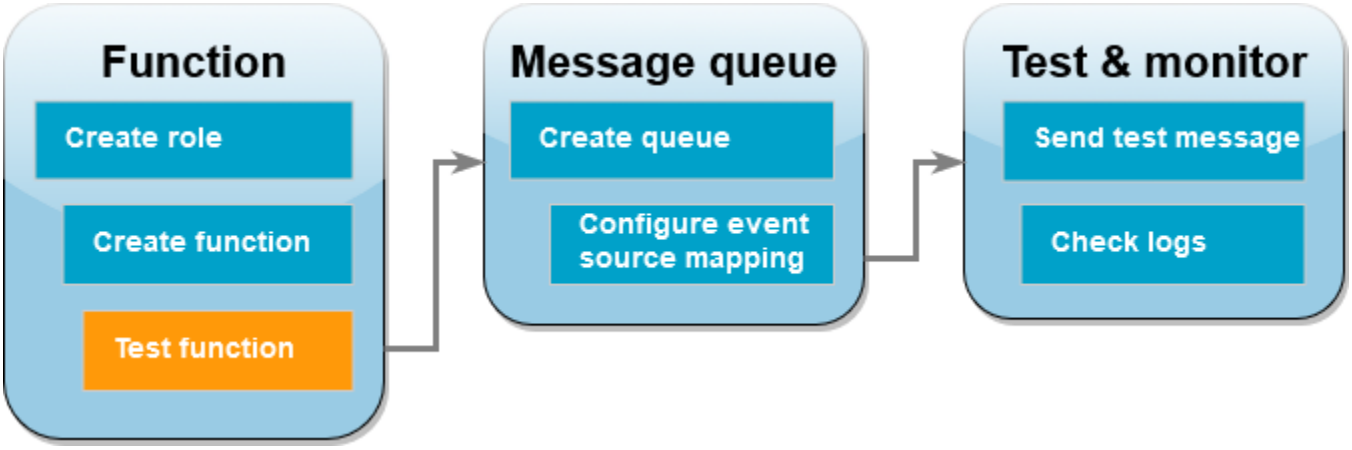
```
zip function.zip index.js
```

4. Créez une fonction Lambda à l'aide de la commande AWS CLI create-function. Pour le paramètre `role`, entrez l'ARN du rôle d'exécution que vous avez créé précédemment.

> **Note**
>
> La fonction Lambda et la file d'attente Amazon SQS doivent se trouver dans la même Région AWS.

```
aws lambda create-function --function-name ProcessSQSRecord \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs24.x \
--role arn:aws:iam::111122223333:role/lambda-sqs-role
```

Tester la fonction



Appelez votre fonction Lambda manuellement à l'aide de la `invoke` AWS CLI commande et d'un exemple d'événement Amazon SQS.

Pour invoquer la fonction Lambda avec un exemple d'événement

1. Enregistrez le JSON suivant en tant que fichier nommé `input.json`. Ce JSON simule un événement qu'Amazon SQS pourrait envoyer à votre fonction Lambda, où `"body"` contient le message réel de la file d'attente. Dans cet exemple, le message est `"test"`.

Example Événement Amazon SQS

Il s'agit d'un événement de test : vous n'avez pas besoin de modifier le message ou le numéro de compte.

```
{
    "Records": [
        {
            "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
            "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
            "body": "test",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082649183",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082649185"
            },
            "messageAttributes": {},
            "md5OfBody": "098f6bcd4621d373cade4e832627b4f6",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-1:111122223333:my-queue",
            "awsRegion": "us-east-1"
        }
    ]
}
```

2. Exécutez la AWS CLI commande d'appel suivante. Cette commande renvoie CloudWatch les journaux dans la réponse. Pour de plus amples informations sur la récupération des journaux, veuillez consulter Accédez aux journaux avec AWS CLI.

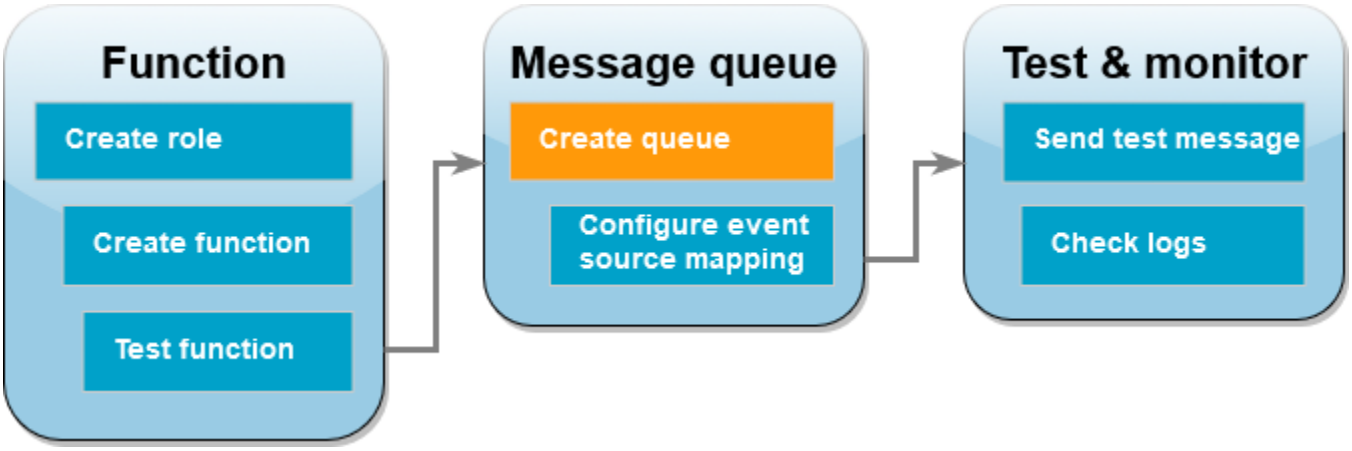
```
aws lambda invoke --function-name ProcessSQSRecord --payload file://input.json out
 --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales prises en charge par l'AWS CLI dans le Guide de l'utilisateur AWS Command Line Interface version 2.

3. Recherchez le journal `INFO` dans la réponse. C'est ici que la fonction Lambda enregistre le corps du message. Vous devriez voir des journaux qui ressemblent à ceci :

```
2023-09-11T22:45:04.271Z 348529ce-2211-4222-9099-59d07d837b60 INFO Processed
 message test
2023-09-11T22:45:04.288Z 348529ce-2211-4222-9099-59d07d837b60 INFO done
```

Créez une file d'attente Amazon SQS.



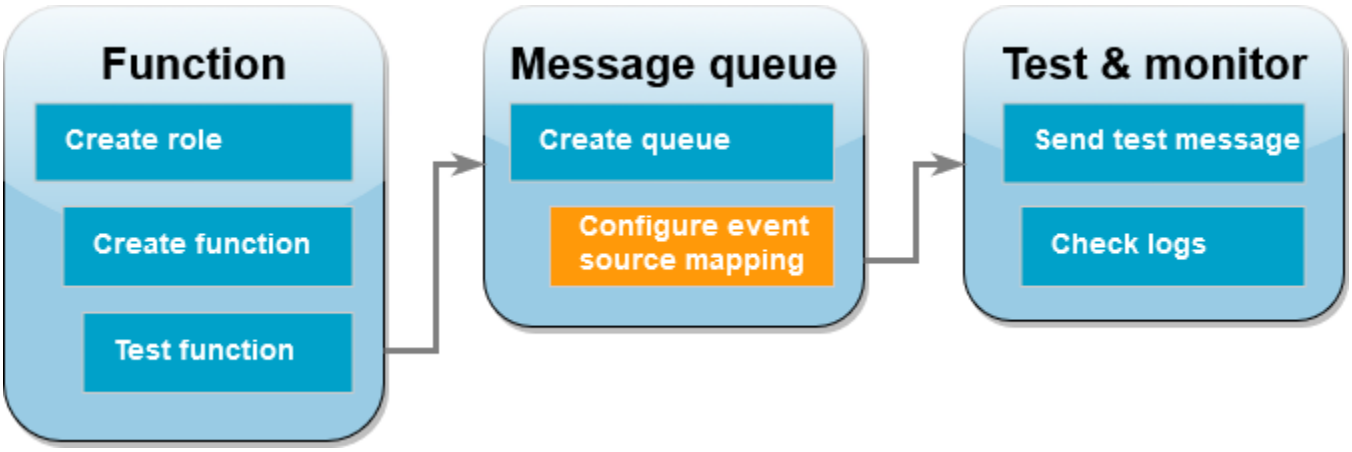
Créez une file d'attente Amazon SQS que la fonction Lambda peut utiliser en tant que source d'événement. La fonction Lambda et la file d'attente Amazon SQS doivent se trouver dans la même Région AWS.

Pour créer une file d'attente

1. Ouvrez la console Amazon SQS.
2. Choisissez Créez une file d'attente.
3. Entrez un nom pour la queue. Conservez les paramètres par défaut de toutes les autres options.
4. Choisissez Créez une file d'attente.

Une fois la file d'attente créée, notez son ARN. Vous en aurez besoin à l'étape suivante lorsque vous associerez la file d'attente à votre fonction Lambda.

Configurer la source de l'événement



Connectez la file d'attente Amazon SQS à votre fonction Lambda en créant un mappage des sources d'événements. Le mappage des sources d'événements lit la file d'attente Amazon SQS et invoque votre fonction Lambda lorsqu'un nouveau message est ajouté.

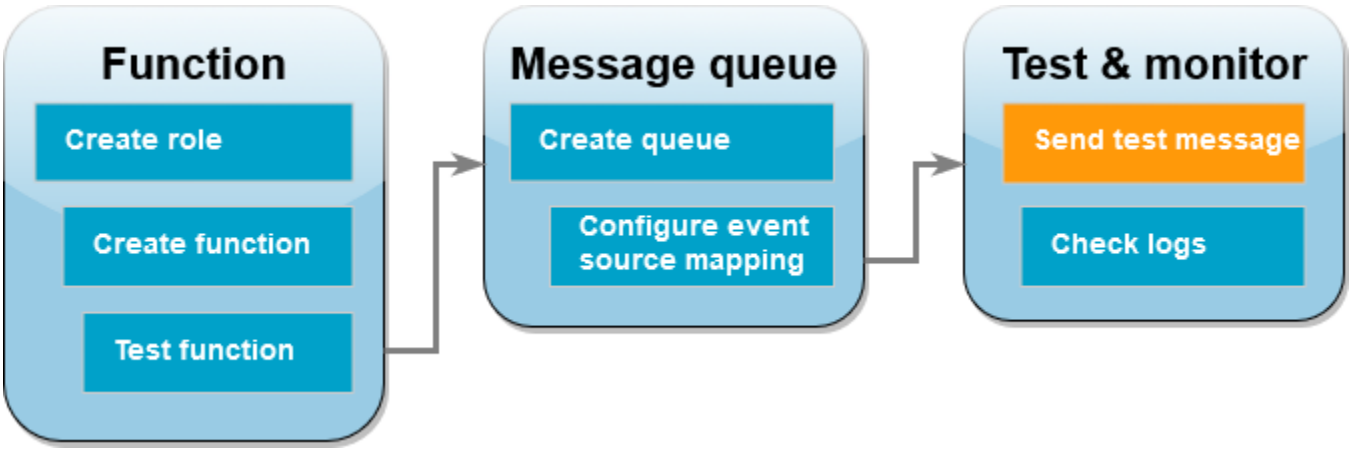
Pour créer un mappage entre votre file d'attente Amazon SQS et votre fonction Lambda, utilisez la commande. create-event-source-mapping AWS CLI Exemple :

```
aws lambda create-event-source-mapping --function-name ProcessSQSRecord  --batch-size
 10 \
--event-source-arn arn:aws:sqs:us-east-1:111122223333:my-queue
```

Pour obtenir la liste de vos mappages de sources d'événements, utilisez la list-event-source-mappingscommande. Exemple :

```
aws lambda list-event-source-mappings --function-name ProcessSQSRecord
```

Envoyer un message de test

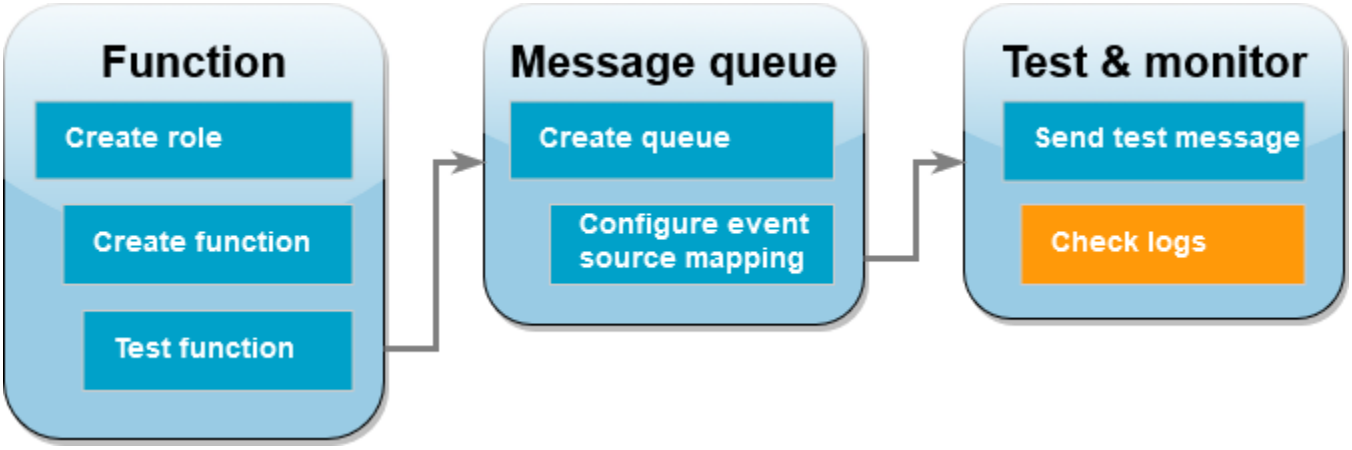


Pour envoyer un message Amazon SQS à la fonction Lambda

1. Ouvrez la console Amazon SQS.
2. Choisissez la queue que vous avez créée précédemment.
3. Choisissez Envoyer et recevoir des messages.
4. Sous Corps du message, entrez un message de test, tel que « ceci est un message de test ».
5. Choisissez Send Message (Envoyer un message).

Lambda interroge la file d'attente concernant les mises à jour. Lorsqu'il y a un nouveau message, Lambda invoque votre fonction avec ces nouvelles données d'événement de la file d'attente. Si le gestionnaire de la fonction revient sans exception, Lambda considère le message comme traité avec succès et commence à lire de nouveaux messages dans la file d'attente. Après avoir traité un message avec succès, Lambda le supprime automatiquement de la file d'attente. Si le gestionnaire renvoie une exception, Lambda considère que le traitement du lot de messages a échoué et invoque la fonction avec le même lot de messages.

Consultez les CloudWatch journaux



Pour confirmer que la fonction a traité le message

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez la SQSRecord fonction Process.
3. Sélectionnez Monitor (Surveiller).
4. Choisissez Afficher CloudWatch les journaux.
5. Dans la CloudWatch console, choisissez le flux de log pour la fonction.
6. Recherchez le journal `INFO`. C'est ici que la fonction Lambda enregistre le corps du message. Vous devriez voir le message que vous avez envoyé depuis la file d'attente Amazon SQS. Exemple :

```
2023-09-11T22:49:12.730Z b0c41e9c-0556-5a8b-af83-43e59efeec71 INFO Processed
 message this is a test message.
```

Nettoyage de vos ressources

Vous pouvez maintenant supprimer les ressources que vous avez créées pour ce didacticiel, sauf si vous souhaitez les conserver. En supprimant AWS les ressources que vous n'utilisez plus, vous évitez des frais inutiles pour votre Compte AWS.

Pour supprimer le rôle d'exécution

1. Ouvrez la page Roles (Rôles) de la console IAM.
2. Sélectionnez le rôle d'exécution que vous avez créé.

3. Sélectionnez Delete (Supprimer).
4. Saisissez le nom du rôle dans le champ de saisie de texte et choisissez Delete (Supprimer).

Pour supprimer la fonction Lambda

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction que vous avez créée.
3. Sélectionnez Actions, Supprimer.
4. Saisissez **`confirm`** dans la zone de saisie de texte et choisissez Delete (Supprimer).

Pour supprimer la file d'attente Amazon SQS

1. Connectez-vous à la console Amazon SQS AWS Management Console et ouvrez-la à l'adresse. https://console.aws.amazon.com/sqs/
2. Sélectionnez la file d'attente que vous avez créée.
3. Sélectionnez Delete (Supprimer).
4. Saisissez **`confirm`** dans le champ de saisie de texte.
5. Sélectionnez Supprimer.

Didacticiel : utilisation d'une file d'attente Amazon SQS entre comptes en tant que source d'événement

Dans ce didacticiel, vous allez créer une fonction Lambda qui consomme les messages d'une file d'attente Amazon Simple Queue Service (Amazon SQS) d'un autre compte. AWS Ce didacticiel implique deux AWS comptes : le compte A fait référence au compte qui contient votre fonction Lambda, et le compte B fait référence au compte qui contient la file d'attente Amazon SQS.

Conditions préalables

Installez le AWS Command Line Interface

Si vous ne l'avez pas encore installé AWS Command Line Interface, suivez les étapes décrites dans la section Installation ou mise à jour de la dernière version du AWS CLI pour l'installer.

Ce tutoriel nécessite un terminal de ligne de commande ou un shell pour exécuter les commandes. Sous Linux et macOS, utilisez votre gestionnaire de shell et de package préféré.

> **Note**
>
> Sous Windows, certaines commandes CLI Bash que vous utilisez couramment avec Lambda (par exemple `zip`) ne sont pas prises en charge par les terminaux intégrés du système d'exploitation. Installez le sous-système Windows pour Linux afin d'obtenir une version intégrée à Windows d'Ubuntu et Bash.

Création du rôle d'exécution (compte A)

Dans le compte A, créez un rôle d'exécution qui autorise votre fonction à accéder aux AWS ressources requises.

Pour créer un rôle d'exécution

1. Ouvrez la page Rôles dans la console Gestion des identités et des accès AWS (IAM).
2. Choisissez Create role (Créer un rôle).
3. Créez un rôle avec les propriétés suivantes :
   - Trusted entity (Entité de confiance) – AWS Lambda.
   - Permissions (Autorisations – AWSLambdaSQSQueueExecutionRole
   - Nom de rôle – **`cross-account-lambda-sqs-role`**

La AWSLambdaSQSQueueExecutionRolepolitique dispose des autorisations dont la fonction a besoin pour lire des éléments depuis Amazon SQS et pour écrire des journaux dans Amazon CloudWatch Logs.

Créer la fonction (compte A)

Dans le Compte A, créez une fonction Lambda qui traite vos messages Amazon SQS. La fonction Lambda et la file d'attente Amazon SQS doivent se trouver dans la même Région AWS.

L'exemple de code Node.js suivant écrit chaque message dans CloudWatch un journal de connexion.

Example index.mjs

```
export const handler = async function(event, context) {
  event.Records.forEach(record => {
```

```
    const { body } = record;
    console.log(body);
  });
  return {};
}
```

Pour créer la fonction

> **Note**
>
> Suivre ces étapes permet de créer une fonction Node.js. Pour les autres langages, les étapes sont similaires mais certains détails sont différents.

1. Enregistrez l'exemple de code en tant que fichier nommé `index.mjs`.
2. Créez un package de déploiement.

   ```
   zip function.zip index.mjs
   ```

3. Créez la fonction à l'aide de la commande `create-function` AWS Command Line Interface (AWS CLI). Remplacez `arn:aws:iam::111122223333:role/cross-account-lambda-sqs-role` par l'ARN du rôle d'exécution que vous avez créé auparavant.

   ```
   aws lambda create-function --function-name CrossAccountSQSExample \
   --zip-file fileb://function.zip --handler index.handler --runtime nodejs24.x \
   --role arn:aws:iam::111122223333:role/cross-account-lambda-sqs-role
   ```

Testez la fonction (compte A)

Dans le compte A, testez votre fonction Lambda manuellement à l'aide de la `invoke` AWS CLI commande et d'un exemple d'événement Amazon SQS.

Si le gestionnaire revient normalement sans exception, Lambda considère le message comme traité avec succès et commence à lire de nouveaux messages dans la file d'attente. Après avoir traité un message avec succès, Lambda le supprime automatiquement de la file d'attente. Si le gestionnaire renvoie une exception, Lambda considère que le traitement du lot de messages a échoué et invoque la fonction avec le même lot de messages.

1. Enregistrez le JSON suivant en tant que fichier nommé `input.txt`.

```
{
    "Records": [
        {
            "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
            "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
            "body": "test",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082649183",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082649185"
            },
            "messageAttributes": {},
            "md5OfBody": "098f6bcd4621d373cade4e832627b4f6",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-1:111122223333:example-queue",
            "awsRegion": "us-east-1"
        }
    ]
}
```

Le JSON précédent simule un événement qu'Amazon SQS pourrait envoyer à votre fonction Lambda, où `"body"` contient le message réel de la file d'attente.

2. Exécutez la commande suivante `invoke` AWS CLI .

```
aws lambda invoke --function-name CrossAccountSQSExample \
--cli-binary-format raw-in-base64-out \
--payload file://input.txt outputfile.txt
```

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

3. Vérifiez la sortie dans le fichier `outputfile.txt`.

Créer une file d'attente Amazon SQS (Compte B)

Dans Compte B, créez une file d'attente Amazon SQS que la fonction Lambda dans Compte B peut utiliser comme source d'événement. La fonction Lambda et la file d'attente Amazon SQS doivent se trouver dans la même Région AWS.

Pour créer une file d'attente

1. Ouvrez la console Amazon SQS.
2. Choisissez Créez une file d'attente.
3. Créez une fille d'attente avec les propriétés suivantes.
   - Type–Standard
   - Nom – LambdaCrossAccountQueue
   - Configuration— Conservez les paramètres par défaut.
   - Stratégie d'accès – Choisissez Avancé. Collez dans la stratégie JSON suivante. Remplacez les valeurs suivantes :
     - 111122223333: Compte AWS ID du compte A
     - 444455556666: Compte AWS ID du compte B

     JSON

```
{
    "Version":"2012-10-17",
    "Id": "Queue1_Policy_UUID",
    "Statement": [
        {
            "Sid": "Queue1_AllActions",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "arn:aws:iam::111122223333:role/cross-account-lambda-
sqs-role"
                ]
            },
            "Action": "sqs:*",
            "Resource": "arn:aws:sqs:us-
east-1:444455556666:LambdaCrossAccountQueue"
        }
    ]
```

```
}
```

Cette stratégie accorde au rôle d'exécution Lambda dansCompte Aautorisations pour consommer des messages de cette file d'attente Amazon SQS.

4. Après avoir créé la file d'attente, enregistrez son Amazon Resource Name (ARN). Vous en aurez besoin à l'étape suivante lorsque vous associerez la file d'attente à votre fonction Lambda.

Configurer la source de l'événement (Compte A)

Dans le compte A, créez un mappage de source d'événement entre la file d'attente Amazon SQS du compte B et votre fonction Lambda en exécutant la commande suivante. `create-event-source-mapping` AWS CLI Remplacez `arn:aws:sqs:us-east-1:444455556666:LambdaCrossAccountQueue` par l'ARN de la file d'attente Amazon SQS que vous avez créée à l'étape précédente.

```
aws lambda create-event-source-mapping --function-name CrossAccountSQSExample --batch-size 10 \
--event-source-arn arn:aws:sqs:us-east-1:444455556666:LambdaCrossAccountQueue
```

Pour obtenir une liste de vos mappages de source d'événements, exécutez la commande suivante.

```
aws lambda list-event-source-mappings --function-name CrossAccountSQSExample \
--event-source-arn arn:aws:sqs:us-east-1:444455556666:LambdaCrossAccountQueue
```

Tester la configuration

Maintenant, vous pouvez tester la configuration comme suit :

1. DansCompte B, ouvrezConsole Amazon SQS.
2. Choisissez LambdaCrossAccountQueuecelui que vous avez créé précédemment.
3. Choisissez Send and receive messages (Envoyer et recevoir des messages).
4. Sous Message body (Corps du message), saisissez un message test.
5. Choisissez Send Message (Envoyer un message).

Votre fonction Lambda dansCompte Adoit recevoir le message. Lambda continuera d'interroger la file d'attente pour les mises à jour. Lorsqu'il y a un nouveau message, Lambda invoque votre fonction

avec ces nouvelles données d'événement de la file d'attente. Votre fonction s'exécute et crée des journaux sur Amazon CloudWatch. Vous pouvez afficher les journaux dans la console CloudWatch .

Nettoyage de vos ressources

Vous pouvez maintenant supprimer les ressources que vous avez créées pour ce didacticiel, sauf si vous souhaitez les conserver. En supprimant AWS les ressources que vous n'utilisez plus, vous évitez des frais inutiles pour votre Compte AWS.

DansCompte A, nettoyez votre rôle d'exécution et votre fonction Lambda.

Pour supprimer le rôle d'exécution

1. Ouvrez la page Roles (Rôles) de la console IAM.
2. Sélectionnez le rôle d'exécution que vous avez créé.
3. Sélectionnez Delete (Supprimer).
4. Saisissez le nom du rôle dans le champ de saisie de texte et choisissez Delete (Supprimer).

Pour supprimer la fonction Lambda

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction que vous avez créée.
3. Sélectionnez Actions, Supprimer.
4. Saisissez **`confirm`** dans la zone de saisie de texte et choisissez Delete (Supprimer).

DansCompte B, nettoyez la file d'attente Amazon SQS.

Pour supprimer la file d'attente Amazon SQS

1. Connectez-vous à la console Amazon SQS AWS Management Console et ouvrez-la à l'adresse. https://console.aws.amazon.com/sqs/
2. Sélectionnez la file d'attente que vous avez créée.
3. Sélectionnez Delete (Supprimer).
4. Saisissez **`confirm`** dans le champ de saisie de texte.
5. Sélectionnez Supprimer.

Orchestration de fonctions Lambda avec Step Functions

Les fonctions Lambda qui gèrent plusieurs tâches, implémentent une logique de nouvelle tentative ou contiennent une logique de branchement sont des anti-modèles. Nous vous recommandons plutôt d'écrire des fonctions Lambda qui exécutent des tâches uniques et d'utiliser AWS Step Functions pour orchestrer les flux de travail de vos applications.

Par exemple, le traitement d'une commande peut nécessiter la validation des détails de la commande, la vérification des niveaux de stock, le traitement du paiement et la génération d'une facture. Écrivez des fonctions Lambda distinctes pour chaque tâche et utilisez Step Functions pour gérer le flux de travail. Step Functions coordonne le flux de données entre vos fonctions et gère les erreurs à chaque étape. Cette séparation facilite la visualisation, la modification et la maintenance de vos flux de travail à mesure qu'ils se complexifient.

Quand utiliser Step Functions avec Lambda

Les scénarios suivants sont de bons exemples d'utilisation de Step Functions pour orchestrer des applications basées sur Lambda.

- Traitement séquentiel
- Gestion des erreurs complexes
- Flux de travail conditionnels et approbations humaines
- Traitement parallèle

Traitement séquentiel

Le traitement séquentiel se produit lorsqu'une tâche doit être terminée avant que la suivante puisse commencer. Par exemple, dans un système de traitement des commandes, le traitement des paiements ne peut pas commencer tant que la validation de la commande n'est pas terminée, et la génération des factures doit attendre la confirmation du paiement. Écrivez des fonctions Lambda distinctes pour chaque tâche et utilisez Step Functions pour gérer la séquence et le flux de données entre les fonctions.

Exemple d'anti-modèle

Une seule fonction Lambda gère l'ensemble du processus de traitement des commandes en :

- Invoquant d'autres fonctions Lambda en séquence

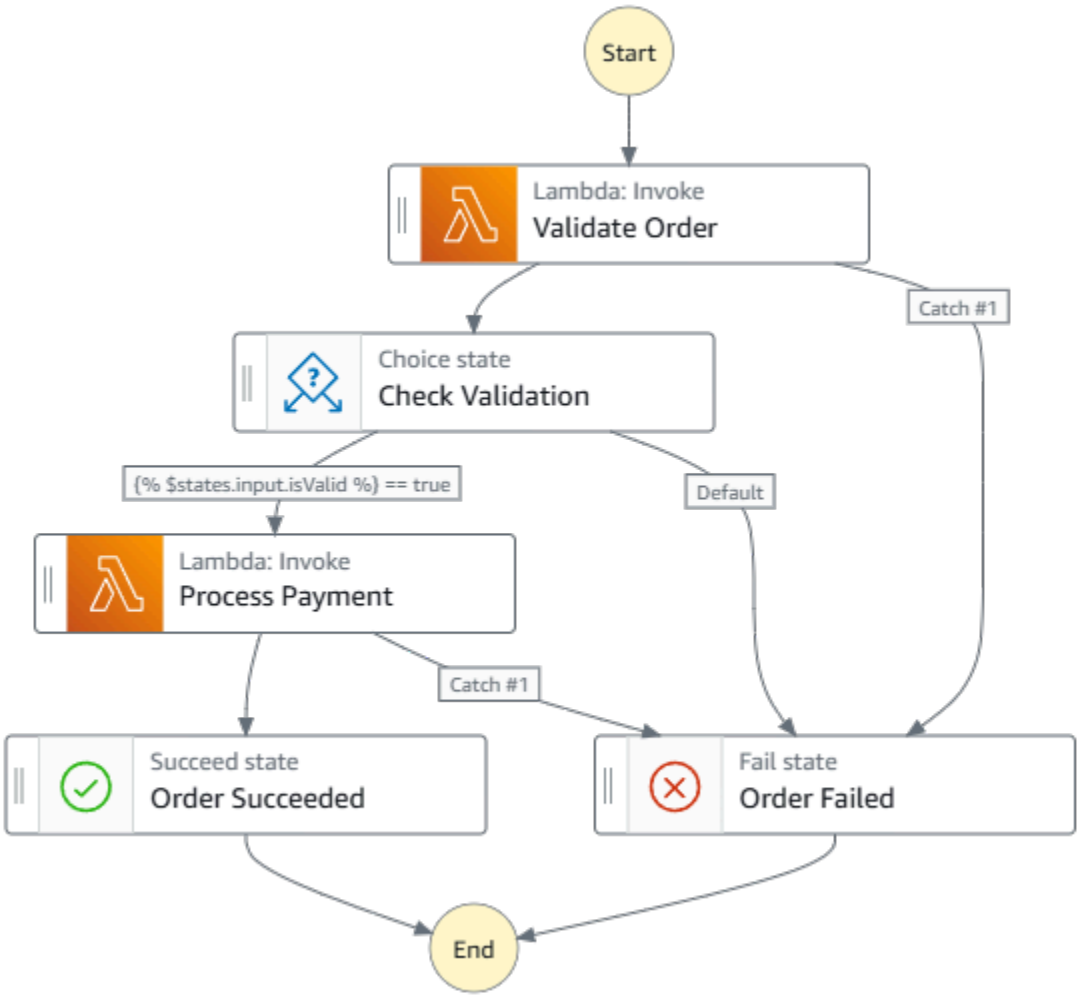
- Analysant et validant les réponses de chaque fonction
- Implémentant la gestion des erreurs et la logique de récupération
- Gérant le flux de données entre les fonctions

Approche recommandée

Utilisez deux fonctions Lambda : l'une pour valider la commande et l'autre pour traiter le paiement. Step Functions coordonne ces fonctions en :

- Exécutant les tâches dans le bon ordre
- Transmettant les données entre les fonctions
- Implémentant la gestion des erreurs à chaque étape
- Utilisant les états Choice pour s'assurer que seules les commandes valides sont réglées

Example graphique de flux de travail



Gestion des erreurs complexes

Bien que Lambda propose des fonctionnalités de nouvelle tentative pour les invocations asynchrones et les mappages des sources d'événements, Step Functions fournit une gestion des erreurs plus sophistiquée pour les flux de travail complexes. Vous pouvez configurer des nouvelles tentatives automatiques avec un backoff exponentiel et définir différentes politiques de tentatives pour différents types d'erreurs. Lorsque les tentatives sont épuisées, utilisez `Catch` pour rediriger les erreurs vers un état de secours. Cela est particulièrement utile lorsque vous avez besoin d'une gestion des erreurs au niveau du flux de travail qui coordonne plusieurs fonctions et services.

Pour en savoir plus sur la gestion des erreurs de fonction Lambda dans une machine à états, consultez Gestion des erreurs dans l'Atelier AWS Step Functions.

Exemple d'anti-modèle

Une seule fonction Lambda gère tout ce qui suit :

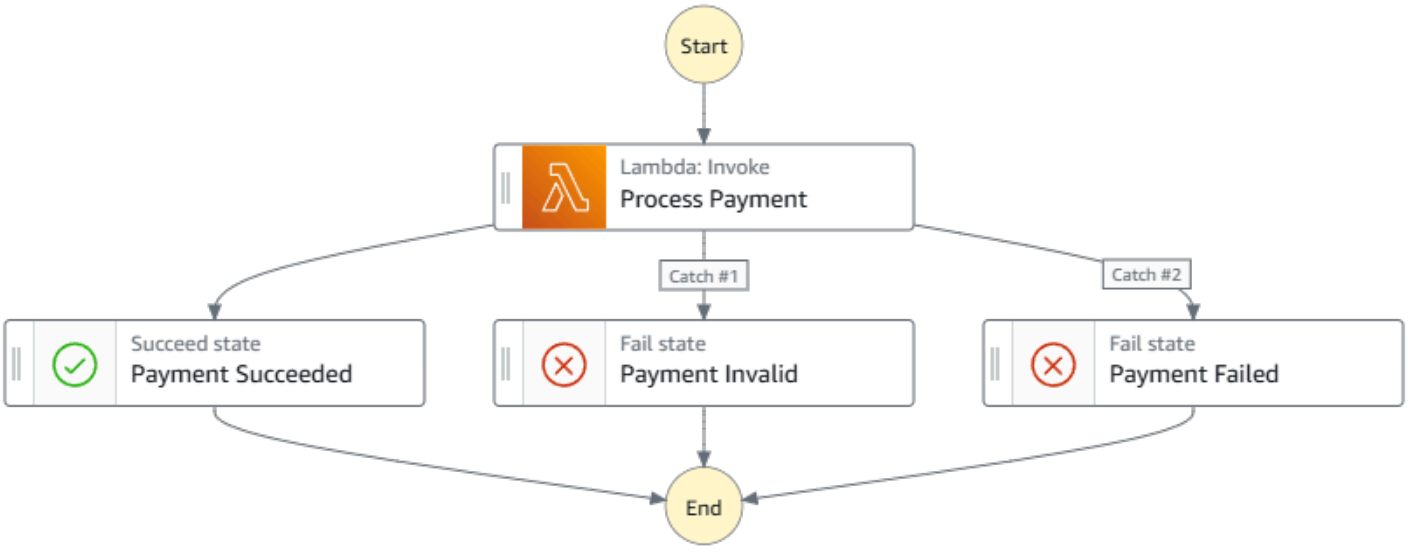
- Tentatives d'appel à un service de traitement des paiements
- Si le service de paiement n'est pas disponible, la fonction attend et essaie à nouveau ultérieurement.
- Implémente un backoff exponentiel personnalisé du temps d'attente
- Quand toutes les tentatives échouent, détecte l'erreur et choisit un autre flux

Approche recommandée

Utilisez une seule fonction Lambda dédiée uniquement au traitement des paiements. Step Functions gère la gestion des erreurs en :

- Réessayant automatiquement les tâches ayant échoué avec des périodes de backoff configurables
- Appliquant différentes politiques de nouvelle tentative en fonction des types d'erreur
- Acheminant différents types d'erreurs vers les états de secours appropriés
- Maintenant l'état et l'historique de gestion des erreurs

Example graphique de flux de travail



Flux de travail conditionnels et approbations humaines

Utilisez l'état Choice Step Functions pour acheminer les flux de travail en fonction du résultat de la fonction et le suffixe waitForTaskToken pour suspendre les flux de travail en fonction de décisions humaines. Par exemple, pour traiter une demande d'augmentation de limite de crédit, utilisez une fonction Lambda pour évaluer les facteurs de risque. Utilisez ensuite Step Functions pour acheminer les demandes à haut risque vers une approbation manuelle et les demandes à faible risque vers une approbation automatique.

Pour déployer un exemple de flux de travail utilisant un modèle d'intégration de jeton de tâche de rappel, consultez Rappel avec jeton de tâche dans l'Atelier AWS Step Functions.

Exemple d'anti-modèle

Une fonction Lambda unique gère un flux de travail d'approbation complexe en :

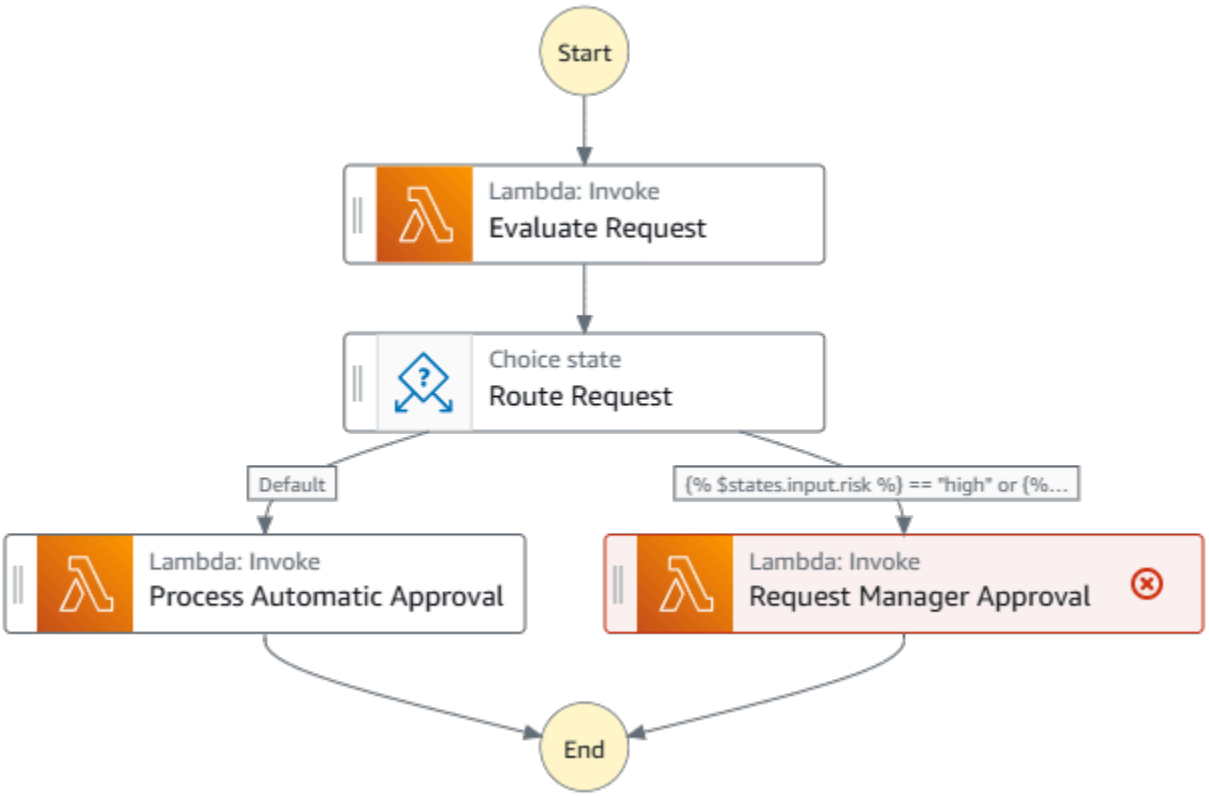
- Implémentant une logique conditionnelle imbriquée pour évaluer les demandes de crédit
- Invoquant différentes fonctions d'approbation en fonction du montant des demandes
- Gérant plusieurs voies d'approbation et points de décision
- Suivant l'état des approbations en attente
- Implémentant une logique de délai d'expiration et de notification pour les approbations

Approche recommandée

Utilisez trois fonctions Lambda : une pour évaluer le risque de chaque demande, une pour approuver les demandes à faible risque et une pour acheminer les demandes à haut risque vers un responsable pour examen. Step Functions gère le flux de travail en :

- Utilisant les états Choice pour acheminer les demandes en fonction du montant et du niveau de risque
- Suspendant l'exécution en attendant l'approbation humaine
- Gérant les délais d'expiration pour les approbations en attente
- Fournissant une visibilité sur l'état actuel de chaque demande

Example graphique de flux de travail



Traitement parallèle

Step Functions propose trois méthodes pour gérer le traitement parallèle :

- L'état Parallel exécute simultanément plusieurs ramifications de votre flux de travail. Utilisez-le lorsque vous devez exécuter différentes fonctions en parallèle, comme la génération de vignettes lors de l'extraction des métadonnées des images.
- L'état Inline Map traite des tableaux de données avec jusqu'à 40 itérations simultanées. Utilisez-le pour les jeux de données de petite ou moyenne taille dans lesquels vous devez effectuer la même opération sur chaque élément.
- L'état Distributed Map gère un traitement parallèle à grande échelle avec jusqu'à 10 000 exécutions simultanées, prenant en charge à la fois les tableaux JSON et les sources de données Amazon Simple Storage Service (Amazon S3). Utilisez-le lorsque vous traitez de grands jeux de données ou lorsque vous avez besoin d'une plus grande simultanéité.

Exemple d'anti-modèle

Une seule fonction Lambda tente de gérer le traitement en parallèle en :

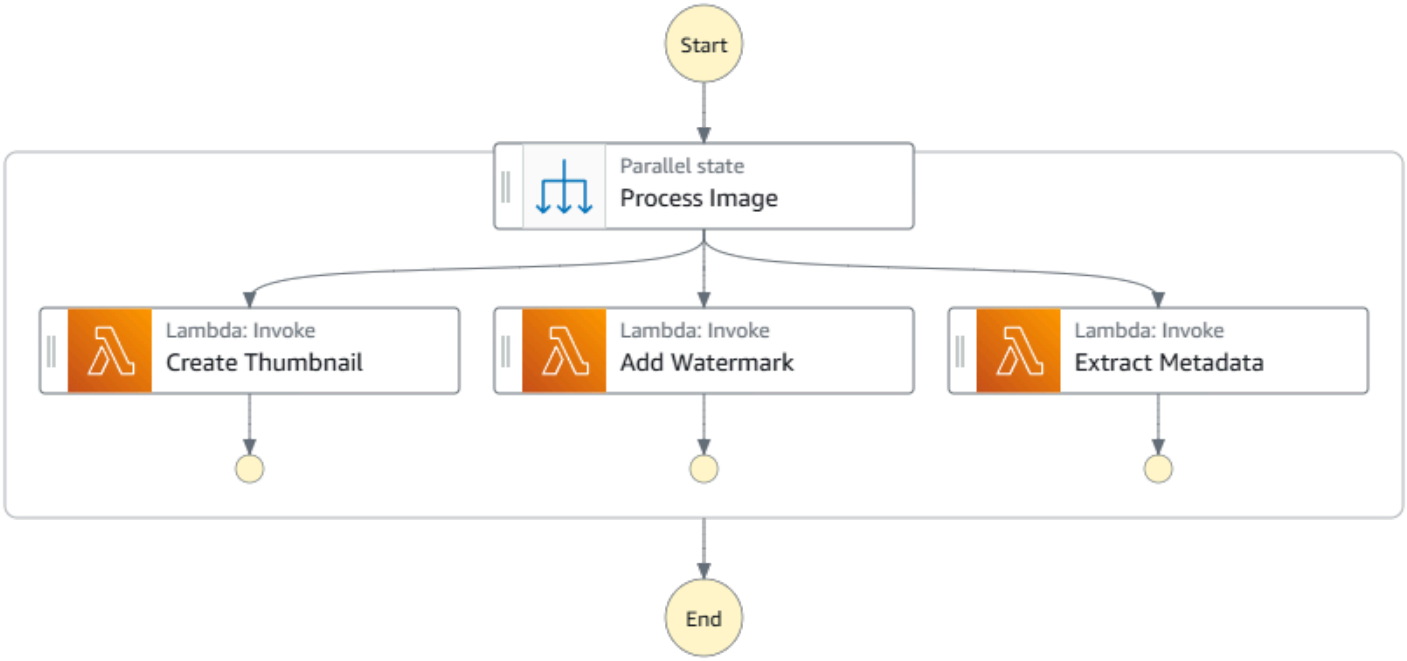
- Invoquant simultanément plusieurs fonctions de traitement d'image
- Implémentant une logique d'exécution parallèle personnalisée
- Gérant les délais d'attente et les erreurs pour chaque tâche parallèle
- Collectant et agrégeant les résultats de toutes les fonctions

Approche recommandée

Utilisez trois fonctions Lambda : une pour créer une vignette, une pour ajouter un filigrane et une pour extraire les métadonnées. Step Functions gère ces fonctions en :

- Exécutant simultanément toutes les fonctions à l'aide de l'état Parallel
- Collectant les résultats de chaque fonction dans un tableau ordonné
- Gérant les délais d'expiration et les erreurs dans toutes les exécutions parallèles
- Continuant uniquement lorsque toutes les ramifications parallèles sont terminées

Example graphique de flux de travail



Quand ne pas utiliser Step Functions avec Lambda

Toutes les applications basées sur Lambda ne bénéficient pas de l'utilisation de Step Functions. Envisagez ces scénarios lorsque vous choisissez l'architecture de votre application.

- Applications simples
- Traitement de données complexes
- Charges de travail intensives en UC

Applications simples

Pour les applications qui ne nécessitent pas d'orchestration complexe, l'utilisation de Step Functions peut ajouter une complexité inutile. Par exemple, si vous traitez simplement des messages provenant d'une file d'attente Amazon SQS ou si vous répondez à des événements Amazon EventBridge, vous pouvez configurer ces services pour invoquer directement vos fonctions Lambda. De même, si votre application ne comprend qu'une ou deux fonctions Lambda avec une gestion simple des erreurs, l'invocation directe de Lambda ou les architectures pilotées par les événements peuvent être plus simples à déployer et gérer.

Traitement de données complexes

Vous pouvez utiliser l'état Distributed Map de Step Functions pour traiter simultanément de grands jeux de données Amazon S3 avec des fonctions Lambda. Cela est efficace pour de nombreuses charges de travail parallèles à grande échelle, notamment pour le traitement de données semi-structurées, comme des fichiers JSON ou CSV. Toutefois, pour des transformations de données plus complexes ou l'analytique avancée, envisagez les alternatives suivantes :

- Pipelines de transformation des données : utilisez AWS Glue pour les tâches ETL qui traitent des données structurées ou semi-structurées provenant de sources multiples. AWS Glue est particulièrement utile lorsque vous avez besoin de fonctionnalités intégrées de gestion de schémas et de catalogues de données.
- Analytique des données : utilisez Amazon EMR pour l'analytique des données à l'échelle du pétaoctet, en particulier lorsque vous avez besoin d'outils de l'écosystème Apache Hadoop ou pour des charges de travail de machine learning qui dépassent les limites de mémoire de Lambda.

Charges de travail intensives en UC

Bien que Step Functions puisse orchestrer des tâches gourmandes en ressources CPU, les fonctions Lambda peuvent ne pas être adaptées à ces charges de travail en raison de leurs ressources CPU limitées. Pour les opérations nécessitant des calculs intensifs au sein de vos flux de travail, envisagez les alternatives suivantes :

- Orchestration de conteneurs : utilisez Step Functions pour gérer les tâches Amazon Elastic Container Service (Amazon ECS) afin d'obtenir des ressources de calcul plus cohérentes et faciles à mettre à l'échelle.
- Traitement par lots : intégrez AWS Batch à Step Functions pour gérer les tâches par lots gourmandes en ressources de calcul qui nécessitent une utilisation soutenue du processeur.

Invocation d'une fonction Lambda à l'aide d'événements par lots Amazon S3

Vous pouvez utiliser des opérations par lot Amazon S3 pour invoquer une fonction Lambda sur un grand ensemble d'objets Amazon S3. Amazon S3 effectue le suivi de la progression des opérations par lot, envoie des notifications et stocke un rapport d'achèvement indiquant le statut de chaque action.

Pour exécuter une opération par lot, vous créez une tâche d'opérations par lot Amazon S3. Lorsque vous créez le travail, vous fournissez un manifeste (la liste des objets) et configurez l'action à effectuer sur ces objets.

Lorsque la tâche par lots démarre, Amazon S3 appelle la fonction Lambda de manière synchrone pour chaque objet figurant dans le manifeste. Le paramètre d'événement inclut les noms du compartiment et de l'objet.

L'exemple suivant illustre l'événement qu'Amazon S3 envoie à la fonction Lambda pour un objet nommé customerImage1.jpg dans le compartiment amzn-s3-demo-bucket.

Example Événement de demande de lot Amazon S3

```
{
"invocationSchemaVersion": "1.0",
    "invocationId": "YXNkbGZqYWRmaiBhc2RmdW9hZHNmZGpmaGFzbGtkaGZza2RmaAo",
    "job": {
        "id": "f3cc4f60-61f6-4a2b-8a21-d07600c373ce"
    },
    "tasks": [
        {
            "taskId": "dGFza2lkZ29lc2hlcmUK",
            "s3Key": "customerImage1.jpg",
            "s3VersionId": "1",
            "s3BucketArn": "arn:aws:s3:::amzn-s3-demo-bucket"
        }
    ]
}
```

Votre fonction Lambda doit renvoyer un objet JSON avec les champs figurant dans l'exemple suivant. Vous pouvez copier `invocationId` et `taskId` à partir du paramètre d'événement. Vous pouvez

renvoyer une chaîne dans `resultString`. Amazon S3 enregistre les valeurs `resultString` dans le rapport de fin de tâche.

Example Réponse à la demande de lot Amazon S3

```
{
  "invocationSchemaVersion": "1.0",
  "treatMissingKeysAs" : "PermanentFailure",
  "invocationId" : "YXNkbGZqYWRmaiBhc2RmdW9hZHNmZGpmaGFzbGtkaGZza2RmaAo",
  "results": [
    {
      "taskId": "dGFza2lkZ29lc2hlcmUK",
      "resultCode": "Succeeded",
      "resultString": "[\"Alice\", \"Bob\"]"
    }
  ]
}
```

Appel de fonctions Lambda à partir d’opérations par lot Amazon S3

Vous pouvez appeler la fonction Lambda avec un ARN de fonction qualifié ou non qualifié. Si vous souhaitez utiliser la même version de fonction pour l’ensemble du travail par lot, configurez une version de fonction spécifique dans le paramètre `FunctionARN` lorsque vous créez votre travail. Si vous configurez un alias ou le qualificateur $LATEST, le travail par lot commence immédiatement à appeler la nouvelle version de la fonction si l’alias ou $LATEST est mis à jour pendant l’exécution du travail.

Notez que vous ne pouvez pas réutiliser une fonction Amazon S3 basée sur un événement existante pour des opérations par lot. En effet, l’opération par lot Amazon S3 transmet un paramètre d’événement différent à la fonction Lambda, et attend un message en retour avec une structure JSON spécifique.

Dans la stratégie basée sur une ressource que vous créez pour la tâche par lot Amazon S3, veillez à définir une autorisation pour que la tâche appelle votre fonction Lambda.

Dans le rôle d’exécution pour la fonction, définissez une stratégie d’approbation pour qu’Amazon S3 endosse ce rôle lors de l’exécution de votre fonction.

Si votre fonction utilise le kit SDK AWS pour gérer les ressources Amazon S3, vous devez ajouter des autorisations Amazon S3 au rôle d'exécution.

Lorsque le travail s'exécute, Amazon S3 démarre plusieurs instances de fonction pour traiter les objets Amazon S3 en parallèle, jusqu'à la limite de simultanéité de la fonction. Amazon S3 limite la montée en puissance initiale des instances afin d'éviter des surcoûts pour les tâches de plus petite taille.

Si la fonction Lambda renvoie un code de réponse `TemporaryFailure`, Amazon S3 réessaie l'opération.

Pour plus d'informations les opérations par lot Amazon S3, consultez Exécution d'opérations par lot dans le Manuel du développeur Amazon S3.

Pour voir un exemple d'utilisation d'une fonction Lambda dans des opérations par lot Amazon S3, consultez Appel d'une fonction Lambda à partir d'opérations par lot Amazon S3 dans le Manuel du développeur Amazon S3.

Invocation des fonctions Lambda à l'aide des notifications Amazon SNS

Utilisez une fonction Lambda pour traiter les notifications Amazon Simple Notification Service (Amazon SNS). Amazon SNS prend en charge les fonctions Lambda en tant que cibles pour les messages envoyés à une rubrique. Vous pouvez abonner votre fonction à des rubriques du même compte ou d'autres comptes AWS. Pour voir une procédure, consultez the section called "didacticiel".

Lambda prend en charge les déclencheurs SNS pour les rubriques SNS standard uniquement. Les rubriques FIFO ne sont pas prises en charge.

Lambda traite les messages SNS de manière asynchrone en les mettant en file d'attente et en gérant les nouvelles tentatives. Si Amazon SNS ne peut pas atteindre Lambda ou si le message est rejeté, Amazon SNS effectue de nouvelles tentatives à intervalles croissants pendant plusieurs heures. Pour plus de détails, consultes Fiabilité dans le FAQ sur Amazon SNS.

> **Warning**
>
> Les invocations asynchrones traitent chaque événement au moins une fois, et le traitement des enregistrements peut être dupliqué. Pour éviter les problèmes potentiels liés à des événements dupliqués, nous vous recommandons vivement de rendre votre code de fonction idempotent. Pour en savoir plus, consultez Comment puis-je rendre ma fonction Lambda idempotente ? dans le Centre de connaissances AWS.

Utilitaire d'idempotence de Powertools pour AWS Lambda

L'utilitaire d'idempotence de Powertools pour AWS Lambda rend vos fonctions Lambda idempotentes. Il est disponible pour Python, TypeScript, Java et .NET. Pour plus d'informations, consultez Utilitaire d'idempotence dans la documentation Powertools pour AWS Lambda (Python), Utilitaire d'idempotence dans la documentation Powertools pour AWS Lambda (TypeScript), Utilitaire d'idempotence dans la documentation Powertools pour AWS Lambda (Java) et Utilitaire d'idempotence dans la documentation Powertools pour AWS Lambda (.NET).

Rubriques

- Ajout d'une rubrique Amazon SNS comme déclencheur de fonction Lambda à l'aide de la console
- Ajout manuel d'une rubrique Amazon SNS comme déclencheur de fonction Lambda

- Exemple de forme d'évènement SNS
- Tutoriel : Utilisation AWS Lambda avec Amazon Simple Notification Service

Ajout d'une rubrique Amazon SNS comme déclencheur de fonction Lambda à l'aide de la console

Pour ajouter une rubrique SNS comme déclencheur d'une fonction Lambda, le moyen le plus simple consiste à utiliser la console Lambda. Lorsque vous ajoutez le déclencheur via la console, Lambda configure automatiquement les autorisations et les abonnements nécessaires pour commencer à recevoir des événements provenant de la rubrique SNS.

Pour ajouter une rubrique SNS comme déclencheur d'une fonction Lambda (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom de la fonction pour laquelle vous souhaitez ajouter le déclencheur.
3. Choisissez Configuration, puis Déclencheurs.
4. Choisissez Add trigger (Ajouter déclencheur).
5. Sous Configuration du déclencheur, dans le menu déroulant, choisissez SNS.
6. Pour Rubrique SNS, choisissez la rubrique SNS à laquelle vous abonner.

Ajout manuel d'une rubrique Amazon SNS comme déclencheur de fonction Lambda

Pour configurer manuellement un déclencheur SNS pour une fonction Lambda, vous devez suivre les étapes suivantes :

- Définir une stratégie basée sur les ressources pour votre fonction afin de permettre à SNS de l'invoquer.
- Abonner votre fonction Lambda à la rubrique Amazon SNS.

> **Note**
>
> Si votre rubrique SNS et votre fonction Lambda se trouvent sur des comptes AWS distincts, vous devez également octroyer des autorisations supplémentaires pour autoriser les

abonnements entre comptes à la rubrique SNS. Pour plus d'informations, consultez Accorder une autorisation multicompte pour l'abonnement Amazon SNS.

Vous pouvez utiliser l'AWS Command Line Interface (AWS CLI) pour effectuer ces deux étapes. Tout d'abord, pour définir une politique basée sur les ressources pour une fonction Lambda qui autorise les invocations SNS, utilisez la commande AWS CLI suivante. Assurez-vous de remplacer la valeur de `--function-name` par le nom de votre fonction Lambda et la valeur de `--source-arn` par l'ARN de votre rubrique SNS.

```
aws lambda add-permission --function-name example-function \
    --source-arn arn:aws:sns:us-east-1:123456789012:sns-topic-for-lambda \
    --statement-id function-with-sns --action "lambda:InvokeFunction" \
    --principal sns.amazonaws.com
```

Pour abonner votre fonction à la rubrique SNS, utilisez la commande AWS CLI suivante. Remplacez la valeur de `--topic-arn` par le nom de votre rubrique ARN et la valeur de `--notification-endpoint` par l'ARN de votre fonction Lambda.

```
aws sns subscribe --protocol lambda \
    --region us-east-1 \
    --topic-arn arn:aws:sns:us-east-1:123456789012:sns-topic-for-lambda \
    --notification-endpoint arn:aws:lambda:us-east-1:123456789012:function:example-
function
```

Exemple de forme d'évènement SNS

Amazon SNS appelle votre fonction de façon asynchrone avec un événement contenant un message et des métadonnées.

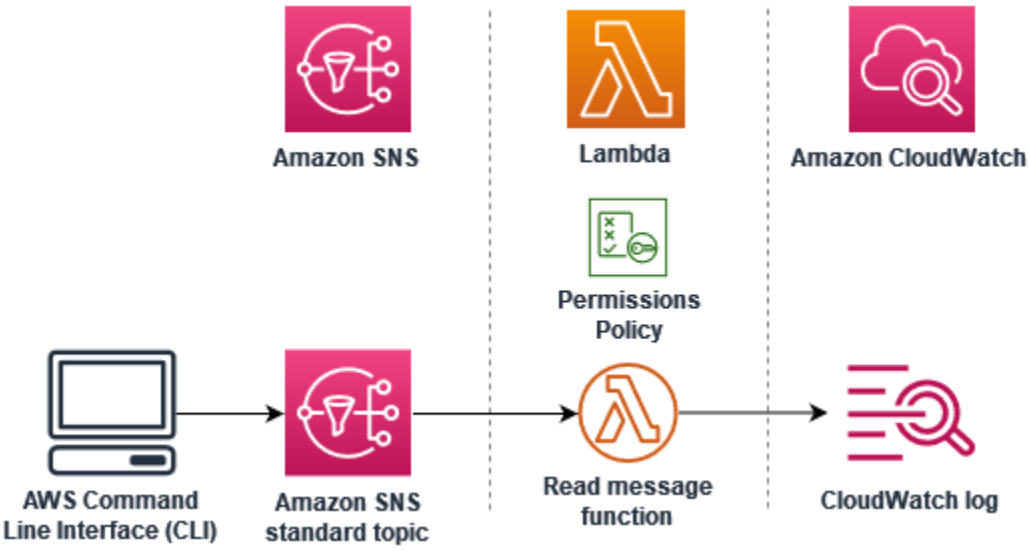
Example Événement de message Amazon SNS

```
{
  "Records": [
    {
      "EventVersion": "1.0",
      "EventSubscriptionArn": "arn:aws:sns:us-east-1:123456789012:sns-lambda:21be56ed-
a058-49f5-8c98-aedd2564c486",
      "EventSource": "aws:sns",
```

```
      "Sns": {
        "SignatureVersion": "1",
        "Timestamp": "2019-01-02T12:45:07.000Z",
        "Signature": "tcc6faL2yUC6dgZdmrwh1Y4cGa/ebXEkAi6RibDsvpi+tE/1+82j...65r==",
        "SigningCertURL": "https://sns.us-east-1.amazonaws.com/
SimpleNotificationService-ac565b8b1a6c5d002d285f9598aa1d9b.pem",
        "MessageId": "95df01b4-ee98-5cb9-9903-4c221d41eb5e",
        "Message": "Hello from SNS!",
        "MessageAttributes": {
          "Test": {
            "Type": "String",
            "Value": "TestString"
          },
          "TestBinary": {
            "Type": "Binary",
            "Value": "TestBinary"
          }
        },
        "Type": "Notification",
        "UnsubscribeUrl": "https://sns.us-east-1.amazonaws.com/?
Action=Unsubscribe&amp;SubscriptionArn=arn:aws:sns:us-east-1:123456789012:test-
lambda:21be56ed-a058-49f5-8c98-aedd2564c486",
        "TopicArn":"arn:aws:sns:us-east-1:123456789012:sns-lambda",
        "Subject": "TestInvoke"
      }
    }
  ]
}
```

Tutoriel : Utilisation AWS Lambda avec Amazon Simple Notification Service

Dans ce didacticiel, vous utiliserez une fonction Lambda intégrée pour vous abonner Compte AWS à une rubrique Amazon Simple Notification Service (Amazon SNS) dans une rubrique distincte. Compte AWS Lorsque vous publiez des messages sur votre rubrique Amazon SNS, votre fonction Lambda lit le contenu du message et le transmet à Amazon Logs. CloudWatch Pour terminer ce didacticiel, vous devez utiliser le AWS Command Line Interface (AWS CLI).



Pour compléter ce tutoriel, effectuez les tâches suivantes :

- Dans le compte A, créez une rubrique Amazon SNS.
- Dans le compte B, créez une fonction Lambda qui lira les messages de la rubrique.
- Dans le compte B, créez un abonnement à la rubrique.
- Publiez des messages sur la rubrique Amazon SNS du compte A et vérifiez que la fonction Lambda du compte B les envoie dans Logs. CloudWatch

En suivant ces étapes, vous apprendrez à configurer une rubrique Amazon SNS pour invoquer une fonction Lambda. Vous apprendrez également comment créer une politique Gestion des identités et des accès AWS (IAM) qui autorise une ressource d'une autre Compte AWS à invoquer Lambda.

Dans le tutoriel, vous utilisez deux Comptes AWS différents. Les AWS CLI commandes illustrent cela en utilisant deux profils nommés appelés `accountA` et`accountB`, chacun étant configuré pour être utilisé avec un profil différent Compte AWS. Pour savoir comment configurer le AWS CLI pour utiliser différents profils, consultez la section Paramètres des fichiers de configuration et d'identification dans le guide de l'AWS Command Line Interface utilisateur de la version 2. Assurez-vous de configurer la même valeur par défaut Région AWS pour les deux profils.

Si les AWS CLI profils que vous créez pour les deux Comptes AWS utilisent des noms différents, ou si vous utilisez le profil par défaut et un profil nommé, modifiez les AWS CLI commandes dans les étapes suivantes selon vos besoins.

Conditions préalables

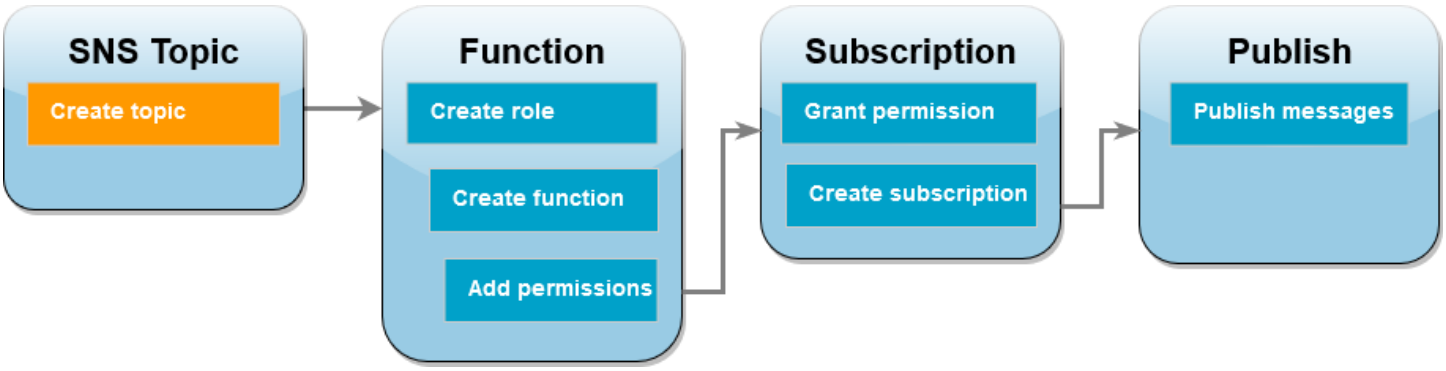
Installez le AWS Command Line Interface

Si vous ne l'avez pas encore installé AWS Command Line Interface, suivez les étapes décrites dans la section Installation ou mise à jour de la dernière version du AWS CLI pour l'installer.

Ce tutoriel nécessite un terminal de ligne de commande ou un shell pour exécuter les commandes. Sous Linux et macOS, utilisez votre gestionnaire de shell et de package préféré.

> **Note**
>
> Sous Windows, certaines commandes CLI Bash que vous utilisez couramment avec Lambda (par exemple `zip`) ne sont pas prises en charge par les terminaux intégrés du système d'exploitation. Installez le sous-système Windows pour Linux afin d'obtenir une version intégrée à Windows d'Ubuntu et Bash.

Créer une rubrique Amazon SNS (compte A)



Pour créer la rubrique

- Dans le compte A, créez une rubrique standard Amazon SNS à l'aide de la commande suivante AWS CLI .

```
aws sns create-topic --name sns-topic-for-lambda --profile accountA
```

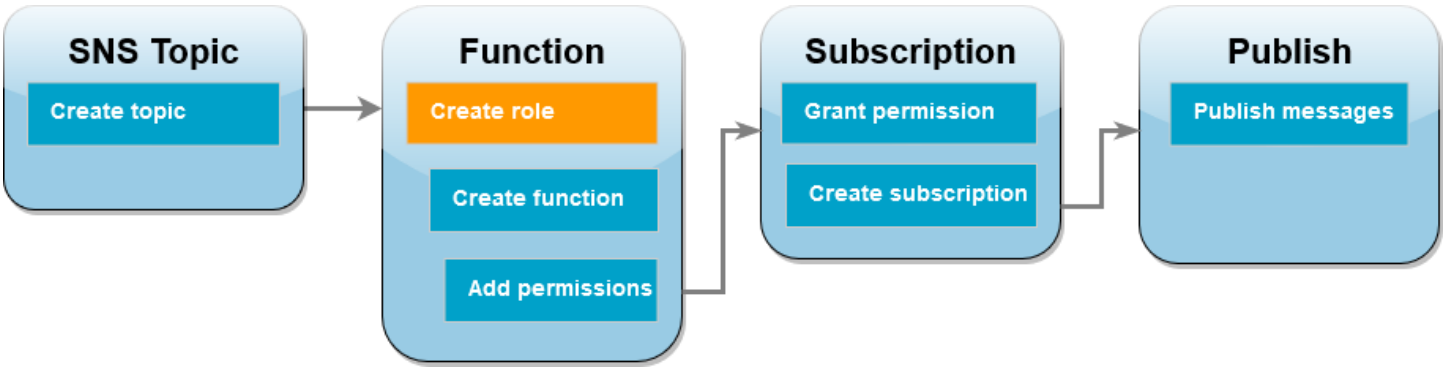
Vous devez visualiser des résultats similaires à ce qui suit.

```
{
    "TopicArn": "arn:aws:sns:us-west-2:123456789012:sns-topic-for-lambda"
```

```
}
```

Notez l'Amazon Resource Name (ARN) de votre rubrique. Vous en aurez besoin plus tard dans le tutoriel lorsque vous ajouterez des autorisations à votre fonction Lambda pour vous abonner à cette rubrique.

Créer un rôle d'exécution de fonction (compte B)



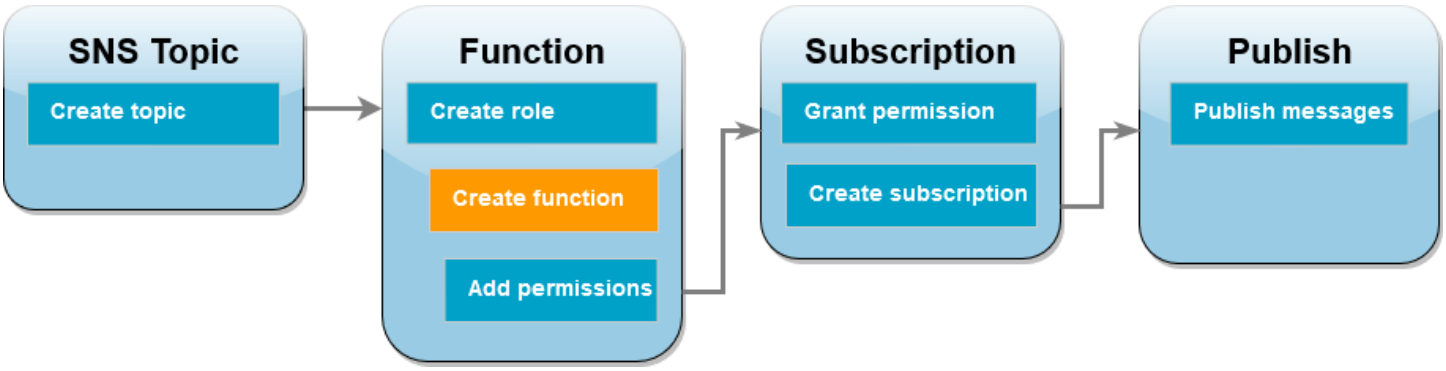
Un rôle d'exécution est un rôle IAM qui accorde à une fonction Lambda l'autorisation d' Services AWS accès et de ressources. Avant de créer votre fonction dans le compte B, vous devez créer un rôle qui donne à la fonction les autorisations de base pour écrire des journaux dans CloudWatch Logs. Nous ajouterons les autorisations de lecture à partir de votre rubrique Amazon SNS à une étape ultérieure.

Pour créer un rôle d'exécution

1. Dans le compte B, ouvrez la page des rôles dans la console IAM.
2. Choisissez Créer un rôle.
3. Pour Type d'entité de confiance, choisissez Service AWS .
4. Pour Cas d'utilisation, choisissez Lambda.
5. Choisissez Suivant.
6. Ajoutez une stratégie d'autorisations de base au rôle en procédant comme suit :
   a. Dans le champ de recherche Politiques d'autorisations, saisissez **`AWSLambdaBasicExecutionRole`**.
   b. Choisissez Suivant.
7. Finalisez la création du rôle en procédant comme suit :
   a. Sous Détails du rôle, saisissez **`lambda-sns-role`** pour Nom du rôle.

b. Choisissez Créer un rôle.

Créer une fonction Lambda (compte B)



Créez une fonction Lambda qui traite vos messages Amazon SNS. Le code de fonction enregistre le contenu des messages de chaque enregistrement dans Amazon CloudWatch Logs.

Ce didacticiel utilise le runtime Node.js 24, mais nous avons également fourni des exemples de code dans d'autres langages d'exécution. Vous pouvez sélectionner l'onglet dans la zone suivante pour voir le code de l'exécution qui vous intéresse. Le JavaScript code que vous allez utiliser dans cette étape se trouve dans le premier exemple présenté dans l'JavaScriptonglet.

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement SNS avec Lambda à l'aide de .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SNSEvents;
```

```
// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SnsIntegration;

public class Function
{
    public async Task FunctionHandler(SNSEvent evnt, ILambdaContext context)
    {
        foreach (var record in evnt.Records)
        {
            await ProcessRecordAsync(record, context);
        }
        context.Logger.LogInformation("done");
    }

    private async Task ProcessRecordAsync(SNSEvent.SNSRecord record,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed record
 {record.Sns.Message}");

            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }
    }
}
```

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement SNS avec Lambda à l'aide de Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, snsEvent events.SNSEvent) {
 for _, record := range snsEvent.Records {
  processMessage(record)
 }
 fmt.Println("done")
}

func processMessage(record events.SNSEventRecord) {
 message := record.SNS.Message
 fmt.Printf("Processed message: %s\n", message)
 // TODO: Process your record here
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement SNS avec Lambda à l'aide de Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SNSEvent;
import com.amazonaws.services.lambda.runtime.events.SNSEvent.SNSRecord;


import java.util.Iterator;
import java.util.List;

public class SNSEventHandler implements RequestHandler<SNSEvent, Boolean> {
    LambdaLogger logger;

    @Override
    public Boolean handleRequest(SNSEvent event, Context context) {
        logger = context.getLogger();
        List<SNSRecord> records = event.getRecords();
        if (!records.isEmpty()) {
            Iterator<SNSRecord> recordsIter = records.iterator();
            while (recordsIter.hasNext()) {
                processRecord(recordsIter.next());
            }
        }
        return Boolean.TRUE;
    }

    public void processRecord(SNSRecord record) {
```

```
        try {
            String message = record.getSNS().getMessage();
            logger.log("message: " + message);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

}
```

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement SNS avec JavaScript Lambda en utilisant.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record) {
  try {
    const message = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
```

```
    throw err;
  }
}
```

Consommation d'un événement SNS avec TypeScript Lambda en utilisant.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SNSEvent, Context, SNSHandler, SNSEventRecord } from "aws-lambda";

export const functionHandler: SNSHandler = async (
  event: SNSEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record: SNSEventRecord): Promise<any> {
  try {
    const message: string = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement SNS avec Lambda à l'aide de PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

/*
Since native PHP support for AWS Lambda is not available, we are utilizing Bref's
 PHP functions runtime for AWS Lambda.
For more information on Bref's PHP runtime for Lambda, refer to: https://bref.sh/
docs/runtimes/function

Another approach would be to create a custom runtime.
A practical example can be found here: https://aws.amazon.com/blogs/apn/aws-
lambda-custom-runtime-for-php-a-practical-example/
*/

// Additional composer packages may be required when using Bref or any other PHP
 functions runtime.
// require __DIR__ . '/vendor/autoload.php';

use Bref\Context\Context;
use Bref\Event\Sns\SnsEvent;
use Bref\Event\Sns\SnsHandler;

class Handler extends SnsHandler
{
    public function handleSns(SnsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $message = $record->getMessage();

            // TODO: Implement your custom processing logic here
            // Any exception thrown will be logged and the invocation will be
 marked as failed

            echo "Processed Message: $message" . PHP_EOL;
        }
    }
}

return new Handler();
```

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement SNS avec Lambda à l'aide de Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for record in event['Records']:
        process_message(record)
    print("done")

def process_message(record):
    try:
        message = record['Sns']['Message']
        print(f"Processed message {message}")
        # TODO; Process your record here

    except Exception as e:
        print("An error occurred")
        raise e
```

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement SNS avec Lambda à l'aide de Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].map { |record| process_message(record) }
end

def process_message(record)
  message = record['Sns']['Message']
  puts("Processing message: #{message}")
rescue StandardError => e
  puts("Error processing message: #{e}")
  raise
end
```

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement S3 avec Lambda à l'aide de Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sns::SnsEvent;
use aws_lambda_events::sns::SnsRecord;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
use tracing::info;

// Built with the following dependencies:
//  aws_lambda_events = { version = "0.10.0", default-features = false, features
 = ["sns"] }
//  lambda_runtime = "0.8.1"
//  tokio = { version = "1", features = ["macros"] }
```

```
//  tracing = { version = "0.1", features = ["log"] }
//  tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }

async fn function_handler(event: LambdaEvent<SnsEvent>) -> Result<(), Error> {
    for event in event.payload.records {
        process_record(&event)?;
    }

    Ok(())
}

fn process_record(record: &SnsRecord) -> Result<(), Error> {
    info!("Processing SNS Message: {}", record.sns.message);

    // Implement your record handling code here.

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

Pour créer la fonction

1. Créez un répertoire pour le projet, puis passez à ce répertoire.

   ```
   mkdir sns-tutorial
   cd sns-tutorial
   ```

2. Copiez l'exemple de JavaScript code dans un nouveau fichier nomméindex.js.
3. Créez un package de déploiement à l'aide de la commande zip suivante.

```
zip function.zip index.js
```

4. Exécutez la AWS CLI commande suivante pour créer votre fonction Lambda dans le compte B.

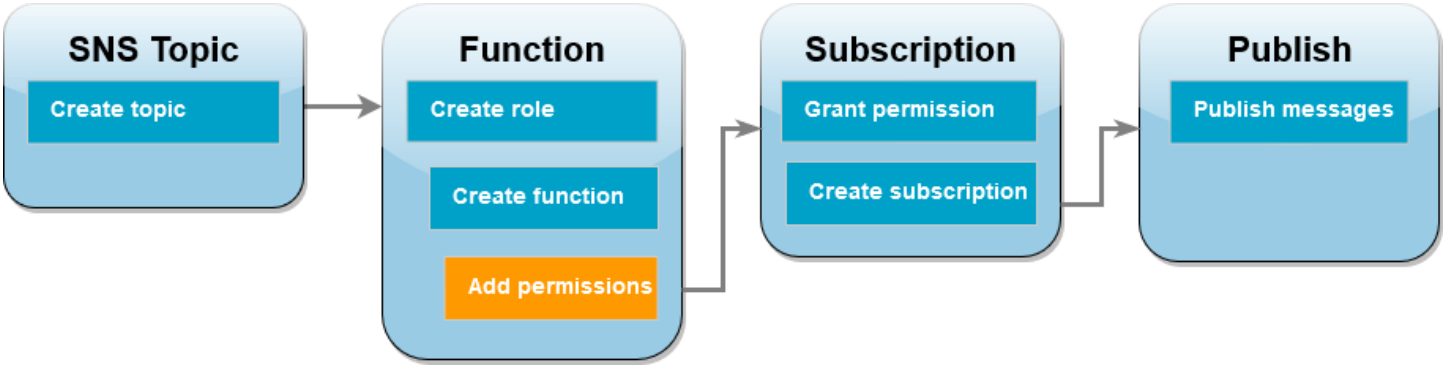
```
aws lambda create-function --function-name Function-With-SNS \
    --zip-file fileb://function.zip --handler index.handler --runtime nodejs24.x \
    --role arn:aws:iam::<AccountB_ID>:role/lambda-sns-role  \
    --timeout 60 --profile accountB
```

   Vous devez visualiser des résultats similaires à ce qui suit.

```
{
    "FunctionName": "Function-With-SNS",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:Function-With-
SNS",
    "Runtime": "nodejs24.x",
    "Role": "arn:aws:iam::123456789012:role/lambda_basic_role",
    "Handler": "index.handler",
    ...
    "RuntimeVersionConfig": {
        "RuntimeVersionArn": "arn:aws:lambda:us-
west-2::runtime:7d5f06b69c951da8a48b926ce280a9daf2e8bb1a74fc4a2672580c787d608206"
    }
}
```

5. Enregistrez l'Amazon Resource Name (ARN) de votre fonction. Vous en aurez besoin plus tard dans le tutoriel lorsque vous ajouterez les autorisations permettant à Amazon SNS d'invoquer votre fonction.

Ajouter des autorisations à la fonction (compte B)



Pour qu'Amazon SNS puisse invoquer votre fonction, vous devez lui accorder une autorisation dans une instruction sur une stratégie basée sur les ressources. Vous ajoutez cette instruction à l'aide de la AWS CLI `add-permission` commande.

Pour accorder à Amazon SNS l'autorisation d'invoquer votre fonction

- Dans le compte B, exécutez la AWS CLI commande suivante en utilisant l'ARN de votre rubrique Amazon SNS que vous avez enregistrée précédemment.

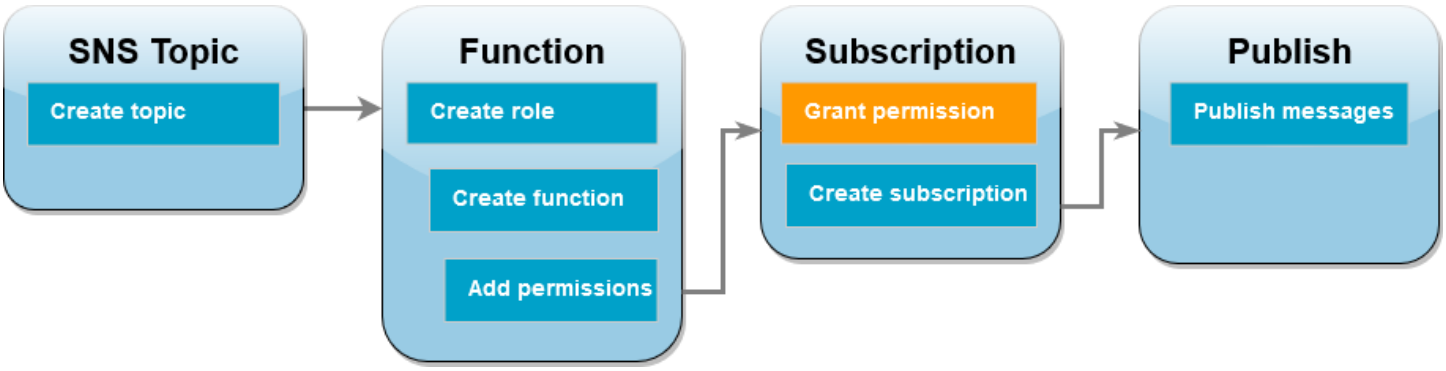
```
aws lambda add-permission --function-name Function-With-SNS \
    --source-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --statement-id function-with-sns --action "lambda:InvokeFunction" \
    --principal sns.amazonaws.com --profile accountB
```

  Vous devez visualiser des résultats similaires à ce qui suit.

```
{
    "Statement": "{\"Condition\":{\"ArnLike\":{\"AWS:SourceArn\":
      \"arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda\"}},
      \"Action\":[\"lambda:InvokeFunction\"],
      \"Resource\":\"arn:aws:lambda:us-east-1:<AccountB_ID>:function:Function-With-
SNS\",
      \"Effect\":\"Allow\",\"Principal\":{\"Service\":\"sns.amazonaws.com\"},
      \"Sid\":\"function-with-sns\"}"
}
```

> **Note**
>
> Si le compte associé à la rubrique Amazon SNS est hébergé dans le cadre d'un opt-in Région AWS, vous devez spécifier la région dans le principal. Si vous travaillez par exemple avec une rubrique Amazon SNS dans la région Asie-Pacifique (Hong Kong), vous devez spécifier `sns.ap-east-1.amazonaws.com` au lieu de `sns.amazonaws.com` pour le principal.

Accorder une autorisation multicompte pour l'abonnement Amazon SNS (compte A)



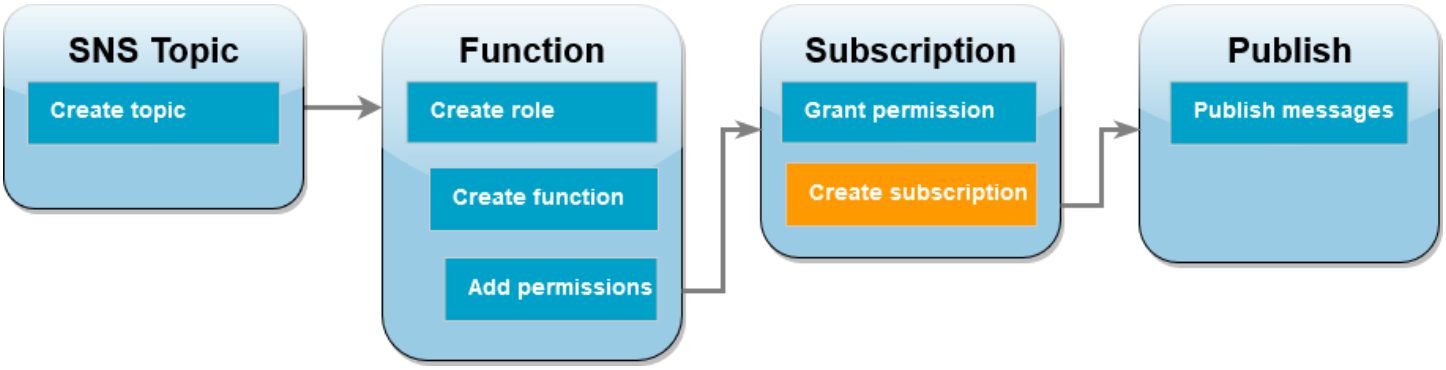
Pour que votre fonction Lambda dans le compte B s'abonne à la rubrique Amazon SNS que vous avez créée dans le compte A, vous devez accorder l'autorisation au compte B de s'abonner à votre rubrique. Vous accordez cette autorisation à l'aide de la AWS CLI `add-permission` commande.

Pour accorder l'autorisation permettant au compte B de s'abonner à la rubrique

- Dans le compte A, exécutez la AWS CLI commande suivante. Utilisez l'ARN pour la rubrique Amazon SNS que vous avez enregistrée précédemment.

```
aws sns add-permission --label lambda-access --aws-account-id <AccountB_ID> \
    --topic-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --action-name Subscribe ListSubscriptionsByTopic --profile accountA
```

Créer un abonnement (compte B)



Dans le compte B, vous abonnez à présent votre fonction Lambda à la rubrique Amazon SNS que vous avez créée au début du tutoriel dans le compte A. Lorsqu'un message est envoyé à cette rubrique (`sns-topic-for-lambda`), Amazon SNS invoque votre fonction Lambda `Function-With-SNS` dans le compte B.

Pour créer un abonnement

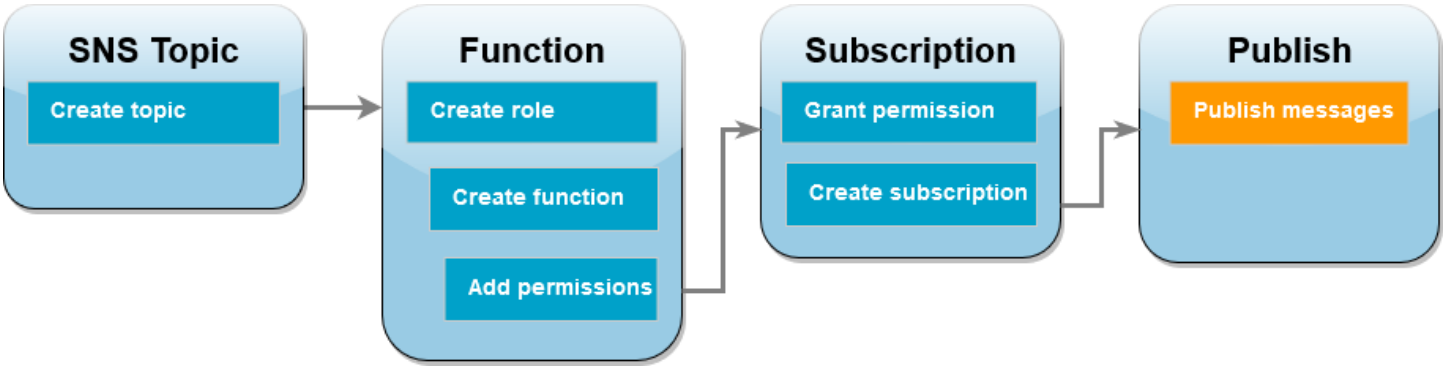
- Dans le compte B, exécutez la AWS CLI commande suivante. Utilisez la région par défaut dans laquelle vous avez créé votre sujet, ainsi que la région ARNs correspondant à votre sujet et la fonction Lambda.

```
aws sns subscribe --protocol lambda \
    --region us-east-1 \
    --topic-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --notification-endpoint arn:aws:lambda:us-
east-1:<AccountB_ID>:function:Function-With-SNS \
    --profile accountB
```

Vous devez visualiser des résultats similaires à ce qui suit.

```
{
    "SubscriptionArn": "arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-
lambda:5d906xxxx-7c8x-45dx-a9dx-0484e31c98xx"
}
```

Publier des messages sur la rubrique (compte A et compte B)



Maintenant que votre fonction Lambda dans le compte B est abonnée à votre rubrique Amazon SNS dans le compte A, il est temps de tester votre configuration en publiant des messages sur votre rubrique. Pour vérifier qu'Amazon SNS a invoqué votre fonction Lambda, utilisez CloudWatch Logs pour afficher le résultat de votre fonction.

Pour publier un message sur votre rubrique et consulter le résultat de votre fonction

1. Saisissez `Hello World` dans un fichier texte et enregistrez-le sous `message.txt`.

2. À partir du répertoire dans lequel vous avez enregistré votre fichier texte, exécutez la AWS CLI commande suivante dans le compte A. Utilisez l'ARN pour votre propre sujet.

```
aws sns publish --message file://message.txt --subject Test \
    --topic-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --profile accountA
```

   Cette commande renverra un ID de message avec un identifiant unique, indiquant qu'Amazon SNS a accepté le message. Amazon SNS tente alors de livrer le message aux abonnés de la rubrique. Pour vérifier qu'Amazon SNS a invoqué votre fonction Lambda, utilisez CloudWatch Logs pour afficher le résultat de votre fonction :

3. Dans le compte B, ouvrez la page Log groups de la CloudWatch console Amazon.
4. Choisissez le groupe de journaux de votre fonction (`/aws/lambda/Function-With-SNS`).
5. Choisissez le flux de journaux le plus récent.
6. Si votre fonction a été correctement invoquée, vous verrez une sortie similaire à la suivante montrant le contenu du message que vous avez publié dans votre rubrique.

```
2023-07-31T21:42:51.250Z c1cba6b8-ade9-4380-aa32-d1a225da0e48 INFO Processed
 message Hello World
2023-07-31T21:42:51.250Z c1cba6b8-ade9-4380-aa32-d1a225da0e48 INFO done
```

Nettoyage de vos ressources

Vous pouvez maintenant supprimer les ressources que vous avez créées pour ce didacticiel, sauf si vous souhaitez les conserver. En supprimant AWS les ressources que vous n'utilisez plus, vous évitez des frais inutiles pour votre Compte AWS.

Dans le compte A, nettoyez votre rubrique Amazon SNS.

Pour supprimer la rubrique Amazon SNS

1. Ouvrez la page Topics (Rubriques) de la console Amazon SNS.
2. Sélectionnez la rubrique que vous avez créée.
3. Choisissez Supprimer.
4. Saisissez **`delete me`** dans le champ de saisie de texte.
5. Sélectionnez Delete (Supprimer).

Dans le compte A, nettoyez votre rôle d'exécution, votre fonction Lambda et votre abonnement Amazon SNS.

Pour supprimer le rôle d'exécution

1. Ouvrez la page Roles (Rôles) de la console IAM.
2. Sélectionnez le rôle d'exécution que vous avez créé.
3. Sélectionnez Delete (Supprimer).
4. Saisissez le nom du rôle dans le champ de saisie de texte et choisissez Delete (Supprimer).

Pour supprimer la fonction Lambda

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction que vous avez créée.
3. Sélectionnez Actions, Supprimer.
4. Saisissez **`confirm`** dans la zone de saisie de texte et choisissez Delete (Supprimer).

Pour supprimer l'abonnement Amazon SNS

1. Ouvrez la page Subscriptions (Abonnements) de la console Amazon SNS.
2. Sélectionnez l'abonnement que vous avez créé.
3. Choisissez Delete (Supprimer), Delete (Supprimer).

Gestion des autorisations dans AWS Lambda

Vous pouvez utiliser Gestion des identités et des accès AWS (IAM) pour gérer les autorisations dans AWS Lambda. Il existe deux catégories principales d'autorisations que vous devez prendre en compte lorsque vous travaillez avec des fonctions Lambda :

- Autorisations dont vos fonctions Lambda ont besoin pour effectuer des actions d'API et accéder à d'autres ressources AWS
- Autorisations dont AWS les autres utilisateurs et entités ont besoin pour accéder à vos fonctions Lambda

Les fonctions Lambda ont souvent besoin d'accéder à d'autres AWS ressources et d'effectuer diverses opérations d'API sur ces ressources. Par exemple, vous pouvez disposer d'une fonction Lambda qui répond à un événement en mettant à jour les entrées d'une base de données Amazon DynamoDB. Dans ce cas, votre fonction a besoin d'autorisations pour accéder à la base de données, ainsi que d'autorisations pour mettre ou mettre à jour des éléments dans cette base de données.

Vous définissez les autorisations dont votre fonction Lambda a besoin dans un rôle IAM spécial appelé rôle d'exécution. Dans ce rôle, vous pouvez associer une politique qui définit toutes les autorisations dont votre fonction a besoin pour accéder à d'autres AWS ressources et lire des sources d'événements. Chaque fonction Lambda doit avoir un rôle d'exécution. Au minimum, votre rôle d'exécution doit avoir accès à Amazon CloudWatch car les fonctions Lambda se connectent par défaut à CloudWatch Logs. Vous pouvez associer la politique gérée par `AWSLambdaBasicExecutionRole` à votre rôle d'exécution pour répondre à cette exigence.

Pour autoriser d'autres Comptes AWS organisations et services à accéder à vos ressources Lambda, plusieurs options s'offrent à vous :

- Vous pouvez utiliser des politiques basées sur l'identité pour accorder à d'autres utilisateurs l'accès à vos ressources Lambda. Les stratégies basées sur l'identité peuvent s'appliquer directement aux utilisateurs, ou aux groupes et aux rôles associés à un utilisateur.
- Vous pouvez utiliser des politiques basées sur les ressources pour accorder à d'autres comptes et Services AWS autorisations l'accès à vos ressources Lambda. Lorsqu'un utilisateur essaie d'accéder à une ressource Lambda, Lambda prend en compte à la fois les stratégies basées sur l'identité appartenant à l'utilisateur et la stratégie basée sur une ressource appartenant aux ressources. Lorsqu'un AWS service tel qu'Amazon Simple Storage Service (Amazon S3) appelle votre fonction Lambda, Lambda ne prend en compte que la politique basée sur les ressources.

- Vous pouvez utiliser un modèle de contrôle d'accès par attributs (ABAC) pour contrôler l'accès à vos fonctions Lambda. Avec ABAC, vous pouvez attacher des balises à une fonction Lambda, les transmettre dans certaines requêtes d'API ou les attacher au principal IAM qui effectue la requête. Spécifiez les mêmes balises dans l'élément condition d'une politique IAM pour contrôler l'accès aux fonctions.

Dans AWS, il est recommandé d'accorder uniquement les autorisations requises pour effectuer une tâche (autorisations du moindre privilège). Pour implémenter cela dans Lambda, nous vous recommandons de commencer par une politique gérée par AWS. Vous pouvez utiliser ces stratégies gérées telles quelles ou comme point de départ pour créer des stratégies plus restrictives.

Pour vous aider à optimiser vos autorisations d'accès avec le moindre privilège, Lambda fournit des conditions supplémentaires que vous pouvez inclure dans vos politiques. Pour de plus amples informations, veuillez consulter the section called "Ressources et conditions".

Pour de plus amples informations sur IAM, consultez le Guide de l'utilisateur IAM.

Définition des autorisations de fonction Lambda avec un rôle d'exécution

Le rôle d'exécution d'une fonction Lambda est un rôle Gestion des identités et des accès AWS (IAM) qui accorde à la fonction l'autorisation d'accès Services AWS et de ressources. Par exemple, vous pouvez créer un rôle d'exécution autorisé à envoyer des journaux à Amazon CloudWatch et à y télécharger des données de suivi AWS X-Ray. Cette page fournit des informations sur la façon de créer, d'afficher et de gérer le rôle d'exécution d'une fonction Lambda.

Lambda assume automatiquement votre rôle d'exécution lorsque vous appelez votre fonction. Vous devez éviter d'appeler `sts:AssumeRole` manuellement afin d'assumer le rôle d'exécution dans votre code de fonction. Si votre cas d'utilisation nécessite que le rôle soit assumé par lui-même, vous devez inclure le rôle lui-même en tant que principal approuvé dans la politique d'approbation de votre rôle. Pour en savoir plus sur la procédure de modification d'une politique d'approbation des rôles, consultez Modification d'une politique d'approbation de rôle (console)dans le Guide de l'utilisateur IAM.

Pour que Lambda assume correctement votre rôle d'exécution, la politique de confiance du rôle doit spécifier le principal de service Lambda (`lambda.amazonaws.com`) comme un service de confiance.

Rubriques

- Création d'un rôle d'exécution dans la console IAM
- Création et gestion des rôles à l'aide du AWS CLI
- Accorder un accès assorti d'un privilège minimum à votre rôle d'exécution Lambda
- Affichage et mise à jour des autorisations dans le rôle d'exécution
- Utilisation de politiques AWS gérées dans le rôle d'exécution
- Utilisation de l'ARN de la fonction source pour contrôler le comportement d'accès aux fonctions

Création d'un rôle d'exécution dans la console IAM

Par défaut, Lambda crée un rôle d'exécution avec des autorisations minimales lorsque vous créez une fonction dans la console Lambda. Plus précisément, ce rôle d'exécution inclut la politique `AWSLambdaBasicExecutionRole` gérée, qui donne à votre fonction les autorisations de base pour consigner des événements sur Amazon CloudWatch Logs.

Vos fonctions ont généralement besoin d'autorisations supplémentaires pour effectuer des tâches plus pertinentes. Par exemple, vous pouvez disposer d'une fonction Lambda qui répond à un événement en mettant à jour les entrées d'une base de données Amazon DynamoDB. Vous pouvez créer un rôle d'exécution avec les autorisations nécessaires à l'aide de la console IAM.

Pour créer un rôle d'exécution dans la console IAM

1. Ouvrez la page Roles (Rôles) dans la console IAM.
2. Choisissez Créer un rôle.
3. Sous Trusted entity type (Type d'entité approuvée), choisissez service AWS .
4. Sous Cas d'utilisation, choisissez Lambda.
5. Choisissez Suivant.
6. Sélectionnez les politiques AWS gérées que vous souhaitez associer à votre rôle. Par exemple, si votre fonction doit accéder à DynamoDB, sélectionnez la politique gérée par DBExecutionDynamo AWSLambda Role.
7. Choisissez Suivant.
8. Entrez un nom dans Role name, puis choisissez Create role.

Pour obtenir des instructions détaillées, consultez la section Création d'un rôle pour un AWS service (console) dans le guide de l'utilisateur IAM.

Après avoir créé votre rôle d'exécution, attachez-le à votre fonction. Lorsque vous créez une fonction dans la console Lambda, vous pouvez attacher n'importe quel rôle d'exécution précédemment créé à la fonction. Si vous souhaitez attacher un nouveau rôle d'exécution à une fonction existante, suivez les étapes décrites dans the section called "Mise à jour du rôle d'exécution d'une fonction".

Création et gestion des rôles à l'aide du AWS CLI

Pour créer un rôle d'exécution avec le AWS Command Line Interface (AWS CLI), utilisez la create-role commande. Lorsque vous utilisez cette commande, vous pouvez préciser la politique d'approbation en ligne. La politique d'approbation d'un rôle accorde aux principaux spécifiés l'autorisation d'assumer le rôle. Dans l'exemple suivant, vous accordez au service Lambda l'autorisation principale d'assumer votre rôle. Notez que les exigences relatives à l'échappement des guillemets dans la chaîne JSON peuvent varier en fonction de votre shell.

```
aws iam create-role \
  --role-name lambda-ex \
```

```
  --assume-role-policy-document '{"Version": "2012-10-17",       "Statement":
[{ "Effect": "Allow", "Principal": {"Service": "lambda.amazonaws.com"}, "Action":
"sts:AssumeRole"}]}'
```

Vous pouvez également définir la politique d'approbation pour le rôle à l'aide d'un fichier JSON séparé. Dans l'exemple suivant, le fichier `trust-policy.json` se trouve dans le répertoire actuel.

Example trust-policy.json

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

```
aws iam create-role \
  --role-name lambda-ex \
  --assume-role-policy-document file://trust-policy.json
```

Pour ajouter des autorisations au rôle, utilisez la commande attach-policy-to-role. La commande suivante ajoute la politique gérée `AWSLambdaBasicExecutionRole` au rôle d'exécution `lambda-ex`.

```
aws iam attach-role-policy --role-name lambda-ex --policy-arn arn:aws:iam::aws:policy/
service-role/AWSLambdaBasicExecutionRole
```

Après avoir créé votre rôle d'exécution, attachez-le à votre fonction. Lorsque vous créez une fonction dans la console Lambda, vous pouvez attacher n'importe quel rôle d'exécution précédemment créé à la fonction. Si vous souhaitez attacher un nouveau rôle d'exécution à une fonction existante, suivez les étapes décrites dans the section called "Mise à jour du rôle d'exécution d'une fonction".

Accorder un accès assorti d'un privilège minimum à votre rôle d'exécution Lambda

Lorsque vous créez pour la première fois un rôle IAM pour votre fonction Lambda pendant la phase de développement, il peut arriver que vous accordiez des autorisations au-delà de ce que est nécessaire. Avant de publier votre fonction dans l'environnement de production, une bonne pratique consiste à ajuster la stratégie de manière à inclure uniquement les autorisations requises. Pour en savoir plus, consultez Appliquer les autorisations de moindre privilège dans le Guide de l'utilisateur IAM.

Utilisez IAM Access Analyzer pour identifier les autorisations requises pour la stratégie de rôle d'exécution IAM. IAM Access Analyzer examine vos AWS CloudTrail journaux pendant la période que vous spécifiez et génère un modèle de politique avec uniquement les autorisations utilisées par la fonction pendant cette période. Vous pouvez utiliser le modèle pour créer une stratégie gérée avec des autorisations affinées, puis l'attacher au rôle IAM. Ainsi, vous accordez uniquement les autorisations dont le rôle a besoin pour interagir avec les AWS ressources correspondant à votre cas d'utilisation spécifique.

Pour en savoir plus, consultez Générer des stratégies basées sur l'activité d'accès dans le Guide de l'utilisateur IAM.

Affichage et mise à jour des autorisations dans le rôle d'exécution

Cette rubrique explique comment afficher et mettre à jour le rôle d'exécution de votre fonction.

Rubriques

- Affichage du rôle d'exécution d'une fonction
- Mise à jour du rôle d'exécution d'une fonction

Affichage du rôle d'exécution d'une fonction

Pour afficher le rôle d'exécution d'une fonction, utilisez la console Lambda.

Pour afficher le rôle d'exécution d'une fonction (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom d'une fonction.
3. Choisissez Configuration (Configuration), puis Permissions (Autorisations).

4. Sous Rôle d'exécution, vous pouvez afficher le rôle actuellement utilisé comme rôle d'exécution de la fonction. Par souci de commodité, vous pouvez consulter toutes les ressources et actions auxquelles la fonction peut accéder dans la section Synthèse des ressources. Vous pouvez également choisir un service dans la liste déroulante pour afficher toutes les autorisations associées à ce service.

Mise à jour du rôle d'exécution d'une fonction

À tout moment, vous pouvez ajouter ou supprimer des autorisations à partir du rôle d'exécution d'une fonction ou configurer votre fonction afin d'utiliser un autre rôle. Si votre fonction a besoin d'accéder à d'autres services ou ressources, vous devez ajouter les autorisations nécessaires au rôle d'exécution.

Lorsque vous ajoutez des autorisations à votre fonction, modifiez légèrement son code ou sa configuration. Cela force l'arrêt et le remplacement des instances en cours d'exécution de votre fonction, qui ont des informations d'identification obsolètes.

Pour mettre à jour le rôle d'exécution d'une fonction, vous pouvez utiliser la console Lambda.

Pour mettre à jour le rôle d'exécution d'une fonction (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom d'une fonction.
3. Choisissez Configuration (Configuration), puis Permissions (Autorisations).
4. Sous Rôle d'exécution, choisissez Modifier.
5. Si vous souhaitez mettre à jour votre fonction pour utiliser un autre rôle en tant que rôle d'exécution, choisissez le nouveau rôle dans le menu déroulant sous Rôle existant.

> **Note**
>
> Si vous souhaitez mettre à jour les autorisations au sein d'un rôle d'exécution existant, vous ne pouvez le faire que dans la console Gestion des identités et des accès AWS (IAM).

Si vous souhaitez créer un nouveau rôle à utiliser comme rôle d'exécution, choisissez Créer un nouveau rôle à partir de modèles de AWS politique sous Rôle d'exécution. Saisissez ensuite

le nom de votre nouveau rôle sous Nom du rôle et spécifiez les politiques que vous souhaitez attacher au nouveau rôle sous Modèles de stratégies.

6. Choisissez Enregistrer.

Utilisation de politiques AWS gérées dans le rôle d'exécution

Les politiques AWS gérées suivantes fournissent les autorisations requises pour utiliser les fonctionnalités Lambda.

Modifier	Description	Date
AWSLambdaMSKExecutionRôle — Lambda a ajouté l'autorisation Kafka : DescribeCluster V2 à cette politique.	`AWSLambdaMSKExecutionRole` accorde les autorisations nécessaires pour lire et accéder aux enregistrements d'un cluster Amazon Managed Streaming for Apache Kafka (Amazon MSK), gérer les interfaces réseau élastiques (ENIs) et écrire dans les journaux. CloudWatch	17 juin 2022
AWSLambdaBasicExecutionRole— Lambda a commencé à suivre les modifications apportées à cette politique.	`AWSLambdaBasicExecutionRole` accorde des autorisations pour charger les journaux sur CloudWatch.	14 février 2022
AWSLambdaDynamoDBExecution Role — Lambda a commencé à suivre les modifications apportées à cette politique.	`AWSLambdaDynamoDBExecutionRole` accorde l'autorisation de lire les enregistrements d'un flux Amazon DynamoDB et d'écrire dans Logs. CloudWatch	14 février 2022

Modifier	Description	Date
AWSLambdaKinesisExecutionRole— Lambda a commencé à suivre les modifications apportées à cette politique.	`AWSLambdaKinesisExecutionRole` accorde l'autorisation de lire les événements d'un flux de données Amazon Kinesis et d'écrire dans Logs. CloudWatch	14 février 2022
AWSLambdaMSKExecutionRôle — Lambda a commencé à suivre les modifications apportées à cette politique.	`AWSLambdaMSKExecutionRole` accorde les autorisations nécessaires pour lire et accéder aux enregistrements d'un cluster Amazon Managed Streaming for Apache Kafka (Amazon MSK), gérer les interfaces réseau élastiques (ENIs) et écrire dans les journaux. CloudWatch	14 février 2022
AWSLambdaSQSQueueExecutionRole— Lambda a commencé à suivre les modifications apportées à cette politique.	`AWSLambdaSQSQueueExecutionRole` accorde l'autorisation de lire un message depuis une file d'attente Amazon Simple Queue Service (Amazon SQS) et d'écrire dans Logs. CloudWatch	14 février 2022
AWSLambdaVPCAccessExecutionRole— Lambda a commencé à suivre les modifications apportées à cette politique.	`AWSLambdaVPCAccessExecutionRole` accorde des autorisations pour gérer ENIs au sein d'un Amazon VPC et écrire dans Logs. CloudWatch	14 février 2022

Modifier	Description	Date
AWSXRayDaemonWriteAccess— Lambda a commencé à suivre les modifications apportées à cette politique.	`AWSXRayDaemonWriteAccess` accorde des autorisations pour charger des données de suivi vers X-Ray.	14 février 2022
CloudWatchLambdaInsightsExecutionRolePolicy— Lambda a commencé à suivre les modifications apportées à cette politique.	`CloudWatchLambdaInsightsExecutionRolePolicy` accorde l'autorisation d'écrire des métriques d'exécution dans CloudWatch Lambda Insights.	14 février 2022
AmazonS3 — ObjectLambdaExecutionRolePolicy Lambda a commencé à suivre les modifications apportées à cette politique.	`AmazonS3ObjectLambdaExecutionRolePolicy` accorde les autorisations nécessaires pour interagir avec l'objet Lambda d'Amazon Simple Storage Service (Amazon S3) et pour écrire dans Logs. CloudWatch	14 février 2022

Pour certaines fonctionnalités, la console Lambda tente d'ajouter les autorisations manquantes au rôle d'exécution dans une stratégie gérée par le client. Ces stratégies peuvent être excessivement nombreuses. Afin d'éviter la création de stratégies supplémentaires, ajoutez les stratégies gérées AWS pertinentes au rôle d'exécution avant d'activer les fonctions.

Lorsque vous utilisez un mappage de source d'événement pour appeler votre fonction, Lambda utilise le rôle d'exécution pour lire les données d'événement. Par exemple, un mappage de source d'événement pour Kinesis lit les événements d'un flux de données et les envoie à votre fonction par lots.

Lorsqu'un service endosse un rôle dans votre compte, vous pouvez inclure les clés de contexte de condition globale `aws:SourceAccount` et `aws:SourceArn` dans votre politique d'approbation des rôles afin de limiter l'accès au rôle aux seules demandes générées par les ressources attendues.

Pour plus d'informations, consultez Prévention du problème de l'adjoint confus entre services pour AWS Security Token Service.

Outre les politiques AWS gérées, la console Lambda fournit des modèles permettant de créer une politique personnalisée avec des autorisations pour des cas d'utilisation supplémentaires. Lorsque vous créez une fonction dans la console Lambda, vous pouvez choisir de créer un nouveau rôle d'exécution doté des autorisations d'un ou plusieurs modèles. Ces modèles s'appliquent également automatiquement lorsque vous créez une fonction à partir d'un plan ou lorsque vous configurez les options qui nécessitent l'accès à d'autres services. Des exemples de modèles sont disponibles dans le GitHubréférentiel de ce guide.

Utilisation de l'ARN de la fonction source pour contrôler le comportement d'accès aux fonctions

Il est courant que votre code de fonction Lambda envoie des requêtes API à d'autres Services AWS. Pour effectuer ces requêtes, Lambda génère un ensemble éphémère d'informations d'identification en assumant le rôle d'exécution de votre fonction. Ces informations d'identification sont disponibles en tant que variables d'environnement lors de l'appel de votre fonction. Lorsque vous travaillez avec AWS SDKs, vous n'avez pas besoin de fournir les informations d'identification du SDK directement dans le code. Par défaut, la chaîne de fournisseurs d'informations d'identification vérifie séquentiellement chaque endroit où vous pouvez définir des informations d'identification et sélectionne le premier disponible, généralement les variables d'environnement (`AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY` et `AWS_SESSION_TOKEN`).

Lambda injecte l'ARN de la fonction source dans le contexte des informations d'identification si la demande est une demande d' AWS API provenant de votre environnement d'exécution. Lambda injecte également l'ARN de la fonction source pour les demandes d'API AWS suivantes que Lambda effectue en votre nom en dehors de votre environnement d'exécution :

Service	Action	Raison
CloudWatch Journaux	`CreateLogGroup` , `CreateLogStream` , `PutLogEvents`	Pour stocker les journaux dans un groupe de CloudWatch journaux
X-Ray	`PutTraceSegments`	Pour envoyer des données de traçage à X-Ray

Service	Action	Raison
Amazon EFS	`ClientMount`	Pour connecter votre fonction à un système de fichiers Amazon Elastic File System (Amazon EFS)

Les autres appels d' AWS API que Lambda effectue en dehors de votre environnement d'exécution en votre nom en utilisant le même rôle d'exécution ne contiennent pas l'ARN de la fonction source. Voici des exemples de tels appels d'API en dehors de l'environnement d'exécution :

- Appelle to AWS Key Management Service (AWS KMS) pour chiffrer et déchiffrer automatiquement vos variables d'environnement.
- Appels à Amazon Elastic Compute Cloud (Amazon EC2) pour créer des interfaces réseau élastiques (ENIs) pour une fonction compatible VPC.
- Appels Services AWS, tels qu'Amazon Simple Queue Service (Amazon SQS), pour lire à partir d'une source d'événements configurée en tant que mappage de sources d'événements.

Avec l'ARN de la fonction source dans le contexte des informations d'identification, vous pouvez vérifier si un appel à votre ressource provient du code d'une fonction Lambda spécifique. Pour vérifier cela, utilisez le clé de condition `lambda:SourceFunctionArn` dans une politique IAM basée sur l'identité ou politique de contrôle des services (SCP).

> **Note**
>
> Vous ne pouvez pas utiliser l'élément `lambda:SourceFunctionArn` dans les politiques basées sur les ressources.

Avec cette clé de condition dans vos politiques basées sur l'identité SCPs, vous pouvez également implémenter des contrôles de sécurité pour les actions d'API que votre code de fonction effectue sur d'autres. Services AWS Cela comporte quelques applications de sécurité clés, telles que l'identification de la source d'une fuite d'informations d'identification.

> **Note**
>
> La clé de condition `lambda:SourceFunctionArn` est différente des clés de condition `lambda:FunctionArn` et `aws:SourceArn`. Le clé de condition `lambda:FunctionArn` s'applique uniquement aux Mappages de sources d'événement et aide à définir les fonctions que votre source d'événement peut invoquer. La clé de `aws:SourceArn` condition s'applique uniquement aux politiques dans lesquelles votre fonction Lambda est la ressource cible et permet de définir quelles autres Services AWS ressources peuvent invoquer cette fonction. La clé de `lambda:SourceFunctionArn` condition peut s'appliquer à n'importe quelle politique basée sur l'identité ou à tout SCP afin de définir les fonctions Lambda spécifiques autorisées à effectuer des appels d' AWS API spécifiques vers d'autres ressources.

Pour utiliser `lambda:SourceFunctionArn` dans votre stratégie, incluez-la en tant que condition dans l'un des Opérateurs de condition ARN. La valeur de la clé doit être un ARN valide.

Par exemple, supposons que votre code de fonction Lambda émette un appel `s3:PutObject` qui cible un compartiment Amazon S3 spécifique. Vous pouvez vouloir n'autoriser qu'une fonction Lambda spécifique à avoir un accès `s3:PutObject` à ce compartiment. Dans ce cas, le rôle d'exécution de votre fonction doit être associé à une stratégie qui ressemble à ceci :

Example stratégie accordant à une fonction Lambda spécifique l'accès à une ressource Amazon S3

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ExampleSourceFunctionArn",
            "Effect": "Allow",
            "Action": "s3:PutObject",
            "Resource": "arn:aws:s3:::lambda_bucket/*",
            "Condition": {
                "ArnEquals": {
                    "lambda:SourceFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:source_lambda"
                }
            }
        }
```

```
    ]
}
```

Cette politique permet uniquement un accès `s3:PutObject` si la source est la fonction Lambda avec ARN `arn:aws:lambda:us-east-1:123456789012:function:source_lambda`. Cette politique n'autorise pas un accès `s3:PutObject` à une autre identité d'appel. C'est le cas même si une fonction ou une entité différente émet un appel `s3:PutObject` avec le même rôle d'exécution.

> **Note**
>
> La clé de condition `lambda:SourceFunctionARN` ne prend pas en charge les versions des fonctions Lambda ni les alias de fonction. Si vous utilisez l'ARN pour une version ou un alias de fonction en particulier, votre fonction n'est pas autorisée à effectuer l'action que vous spécifiez. Veillez à utiliser l'ARN non qualifié pour votre fonction sans suffixe de version ou d'alias.

Vous pouvez également l'utiliser `lambda:SourceFunctionArn` dans SCPs. Supposons, par exemple, que vous souhaitiez limiter l'accès à votre compartiment à un code de fonction Lambda unique ou à des appels issus d'un cloud privé virtuel (VPC) Amazon. Le SCP suivant illustre ce processus.

Example politique refusant l'accès à Amazon S3 dans des conditions spécifiques

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "s3:*"
            ],
            "Resource": "arn:aws:s3:::lambda_bucket/*",
            "Effect": "Deny",
            "Condition": {
                "StringNotEqualsIfExists": {
                    "aws:SourceVpc": [
                        "vpc-12345678"
```

```
                    ]
                }
            }
        },
        {
            "Action": [
                "s3:*"
            ],
            "Resource": "arn:aws:s3:::lambda_bucket/*",
            "Effect": "Deny",
            "Condition": {
                "ArnNotEqualsIfExists": {
                    "lambda:SourceFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:source_lambda"
                }
            }
        }
    ]
}
```

Cette stratégie refuse toutes les actions S3 sauf si elles proviennent d'une fonction Lambda spécifique avec ARN `arn:aws:lambda:*:123456789012:function:source_lambda`, ou à moins qu'ils ne proviennent du VPC spécifié. L'opérateur `StringNotEqualsIfExists` indique à IAM de traiter cette condition uniquement si la clé `aws:SourceVpc` est présente dans la demande. De la même façon, IAM considère l'opérateur `ArnNotEqualsIfExists` uniquement si le `lambda:SourceFunctionArn` existe.

Accorder à d'autres AWS entités l'accès à vos fonctions Lambda

Pour autoriser d'autres Comptes AWS organisations et services à accéder à vos ressources Lambda, plusieurs options s'offrent à vous :

- Vous pouvez utiliser des politiques basées sur l’identité pour accorder à d’autres utilisateurs l’accès à vos ressources Lambda. Les stratégies basées sur l’identité peuvent s’appliquer directement aux utilisateurs, ou aux groupes et aux rôles associés à un utilisateur.
- Vous pouvez utiliser des politiques basées sur les ressources pour accorder à d'autres comptes et Services AWS autorisations l'accès à vos ressources Lambda. Lorsqu’un utilisateur essaie d’accéder à une ressource Lambda, Lambda prend en compte à la fois les stratégies basées sur l’identité appartenant à l’utilisateur et la stratégie basée sur une ressource appartenant aux ressources. Lorsqu'un AWS service tel qu'Amazon Simple Storage Service (Amazon S3) appelle votre fonction Lambda, Lambda ne prend en compte que la politique basée sur les ressources.
- Vous pouvez utiliser un modèle de contrôle d’accès par attributs (ABAC) pour contrôler l’accès à vos fonctions Lambda. Avec ABAC, vous pouvez attacher des balises à une fonction Lambda, les transmettre dans certaines requêtes d’API ou les attacher au principal IAM qui effectue la requête. Spécifiez les mêmes balises dans l’élément condition d’une politique IAM pour contrôler l’accès aux fonctions.

Pour vous aider à optimiser vos autorisations d’accès avec le moindre privilège, Lambda fournit des conditions supplémentaires que vous pouvez inclure dans vos politiques. Pour de plus amples informations, veuillez consulter the section called “Ressources et conditions”.

Politiques IAM basées sur l’identité pour Lambda

Vous pouvez utiliser des politiques basées sur l'identité dans Gestion des identités et des accès AWS (IAM) pour autoriser les utilisateurs de votre compte à accéder à Lambda. Les politiques basées sur l’identité peuvent s’appliquer aux utilisateurs, aux groupes d’utilisateurs ou aux rôles. Vous pouvez également accorder aux utilisateurs d’un autre compte l’autorisation d’endosser un rôle de votre compte et d’accéder à vos ressources Lambda.

Lambda fournit des politiques AWS gérées qui accordent l'accès aux actions de l'API Lambda et, dans certains cas, à d'autres AWS services utilisés pour développer et gérer les ressources Lambda. Lambda met à jour ces stratégies gérées si nécessaire pour s’assurer que vos utilisateurs ont accès aux nouvelles fonctionnalités quand elles sont disponibles.

- AWSLambda_FullAccess— Accorde un accès complet aux actions Lambda et aux autres AWS services utilisés pour développer et gérer les ressources Lambda.
- AWSLambda_ReadOnlyAccess— Accorde un accès en lecture seule aux ressources Lambda.
- AWSLambdaRôle : accorde des autorisations pour appeler des fonctions Lambda.

AWS les politiques gérées autorisent les actions d'API sans restreindre les fonctions ou les couches Lambda qu'un utilisateur peut modifier. Pour bénéficier d'un contrôle plus précis, vous pouvez créer vos propres stratégies qui limitent la portée des autorisations d'un utilisateur.

Rubriques

- Octroi aux utilisateurs de l'accès à une fonction Lambda
- Octroi aux utilisateurs de l'accès à une couche Lambda

Octroi aux utilisateurs de l'accès à une fonction Lambda

Utilisez des politiques basées sur l'identité pour autoriser des utilisateurs, des groupes d'utilisateurs ou des groupes à effectuer des opérations sur des fonctions Lambda.

> **Note**
>
> Pour une fonction définie comme une image de conteneur, l'autorisation des utilisateurs à accéder à l'image doit être configurée dans Amazon Elastic Container Registry (Amazon ECR). Pour voir un exemple, consultez la rubrique Politiques de référentiel Amazon ECR.

Voici un exemple de stratégie d'autorisations avec une portée limitée. Elle permet à un utilisateur de créer et gérer des fonctions Lambda nommées avec un préfixe distinct (`intern-`) et configurées avec un rôle d'exécution désigné.

Example Stratégie de développement des fonctions

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
```

```
            "Sid": "ReadOnlyPermissions",
            "Effect": "Allow",
            "Action": [
                "lambda:GetAccountSettings",
                "lambda:GetEventSourceMapping",
                "lambda:GetFunction",
                "lambda:GetFunctionConfiguration",
                "lambda:GetFunctionCodeSigningConfig",
                "lambda:GetFunctionConcurrency",
                "lambda:ListEventSourceMappings",
                "lambda:ListFunctions",
                "lambda:ListTags",
                "iam:ListRoles"
            ],
            "Resource": "*"
        },
        {
            "Sid": "DevelopFunctions",
            "Effect": "Allow",
            "NotAction": [
                "lambda:AddPermission",
                "lambda:PutFunctionConcurrency"
            ],
            "Resource": "arn:aws:lambda:*:*:function:intern-*"
        },
        {
            "Sid": "DevelopEventSourceMappings",
            "Effect": "Allow",
            "Action": [
                "lambda:DeleteEventSourceMapping",
                "lambda:UpdateEventSourceMapping",
                "lambda:CreateEventSourceMapping"
            ],
            "Resource": "*",
            "Condition": {
                "ArnLike": {
                    "lambda:FunctionArn": "arn:aws:lambda:*:*:function:intern-*"
                }
            }
        },
        {
            "Sid": "PassExecutionRole",
            "Effect": "Allow",
            "Action": [
```

```
                "iam:ListRolePolicies",
                "iam:ListAttachedRolePolicies",
                "iam:GetRole",
                "iam:GetRolePolicy",
                "iam:PassRole",
                "iam:SimulatePrincipalPolicy"
            ],
            "Resource": "arn:aws:iam::*:role/intern-lambda-execution-role"
        },
        {
            "Sid": "ViewLogs",
            "Effect": "Allow",
            "Action": [
                "logs:*"
            ],
            "Resource": "arn:aws:logs:*:*:log-group:/aws/lambda/intern-*"
        }
    ]
}
```

Les autorisations de la stratégie sont organisées en déclarations reposant sur les ressources et conditions qu'elles prennent en charge.

- `ReadOnlyPermissions` – La console Lambda utilise ces autorisations lorsque vous parcourez et affichez les fonctions. Les conditions ou modèles de ressources ne sont pas pris en charge.

```
        "Action": [
            "lambda:GetAccountSettings",
            "lambda:GetEventSourceMapping",
            "lambda:GetFunction",
            "lambda:GetFunctionConfiguration",
            "lambda:GetFunctionCodeSigningConfig",
            "lambda:GetFunctionConcurrency",
            "lambda:ListEventSourceMappings",
            "lambda:ListFunctions",
            "lambda:ListTags",
            "iam:ListRoles"
        ],
        "Resource": "*"
```

- `DevelopFunctions` – Utilisez toute action Lambda qui s'exécute sur des fonctions ayant le préfixe `intern-`, à l'exception de `AddPermission` et de `PutFunctionConcurrency`. `AddPermission` modifie la politique basée sur les ressources de la fonction, ce qui peut avoir des conséquences en matière de sécurité. `PutFunctionConcurrency` réserve la capacité de mise à l'échelle pour une fonction, et peut retirer de la capacité à d'autres fonctions.

```
        "NotAction": [
            "lambda:AddPermission",
            "lambda:PutFunctionConcurrency"
        ],
        "Resource": "arn:aws:lambda:*:*:function:intern-*"
```

- `DevelopEventSourceMappings` – Gérez les mappages de source d'événement sur les fonctions qui sont préfixées avec `intern-`. Ces actions s'exécutent sur des mappages de source d'événement, mais vous pouvez les restreindre par fonction à l'aide d'une condition.

```
        "Action": [
            "lambda:DeleteEventSourceMapping",
            "lambda:UpdateEventSourceMapping",
            "lambda:CreateEventSourceMapping"
        ],
        "Resource": "*",
        "Condition": {
            "StringLike": {
                "lambda:FunctionArn": "arn:aws:lambda:*:*:function:intern-*"
            }
        }
```

- `PassExecutionRole` – Affichez et transmettez uniquement un rôle nommé `intern-lambda-execution-role`, qui doit être créé et géré par un utilisateur avec des autorisations IAM. `PassRole` est utilisé lorsque vous attribuez un rôle d'exécution à une fonction.

```
        "Action": [
            "iam:ListRolePolicies",
            "iam:ListAttachedRolePolicies",
            "iam:GetRole",
            "iam:GetRolePolicy",
            "iam:PassRole",
```

```
            "iam:SimulatePrincipalPolicy"
        ],
        "Resource": "arn:aws:iam::*:role/intern-lambda-execution-role"
```

- `ViewLogs`— Utilisez CloudWatch les journaux pour afficher les journaux des fonctions préfixées par`intern-`.

```
        "Action": [
            "logs:*"
        ],
        "Resource": "arn:aws:logs:*:*:log-group:/aws/lambda/intern-*"
```

Cette stratégie permet à un utilisateur de démarrer avec Lambda, sans nuire aux ressources des autres utilisateurs. Il ne permet pas à un utilisateur de configurer une fonction pour qu'elle soit déclenchée par d'autres services ou d'appeler d'autres AWS services, ce qui nécessite des autorisations IAM plus étendues. Cela n'inclut pas non plus l'autorisation d'accéder à des services qui ne sont pas compatibles avec les politiques à portée limitée, tels que CloudWatch X-Ray. Pour ces services, utilisez les stratégies en lecture seule afin d'accorder à l'utilisateur l'accès aux métriques et aux données de suivi.

Lorsque vous configurez des déclencheurs pour votre fonction, vous devez y accéder pour utiliser le AWS service qui appelle votre fonction. Par exemple, pour configurer un déclencheur Amazon S3, vous vous devez être autorisé à utiliser les actions Amazon S3 qui gèrent les notifications de compartiment. La plupart de ces autorisations sont incluses dans la politique AWSLambda_FullAccessgérée.

Octroi aux utilisateurs de l'accès à une couche Lambda

Utilisez des politiques basées sur l'identité pour autoriser des utilisateurs, des groupes d'utilisateurs ou des groupes à effectuer des opérations sur des couches Lambda. La stratégie suivante accorde à un utilisateur l'autorisation de créer des couches et de les utiliser avec des fonctions. Les modèles de ressources permettent à l'utilisateur de travailler dans n' Région AWS importe quelle version de couche, à condition que le nom de la couche commence par`test-`.

Example stratégie de développement des couches

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "PublishLayers",
            "Effect": "Allow",
            "Action": [
                "lambda:PublishLayerVersion"
            ],
            "Resource": "arn:aws:lambda:*:*:layer:test-*"
        },
        {
            "Sid": "ManageLayerVersions",
            "Effect": "Allow",
            "Action": [
                "lambda:GetLayerVersion",
                "lambda:DeleteLayerVersion"
            ],
            "Resource": "arn:aws:lambda:*:*:layer:test-*:*"
        }
    ]
}
```

Vous pouvez également appliquer l'utilisation des couches durant la création et la configuration de la fonction avec la condition `lambda:Layer`. Par exemple, vous pouvez empêcher les utilisateurs d'utiliser les couches publiées par d'autres comptes. La stratégie ci-dessous ajoute une condition aux actions `CreateFunction` et `UpdateFunctionConfiguration` afin d'exiger que toutes les couches spécifiées proviennent du compte `123456789012`.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ConfigureFunctions",
```

```
            "Effect": "Allow",
            "Action": [
                "lambda:CreateFunction",
                "lambda:UpdateFunctionConfiguration"
            ],
            "Resource": "*",
            "Condition": {
                "ForAllValues:StringLike": {
                    "lambda:Layer": [
                        "arn:aws:lambda:*:123456789012:layer:*:*"
                    ]
                }
            }
        }
    ]
}
```

Pour que la condition s'applique, vérifiez qu'aucune autre déclaration n'accorde d'autorisation utilisateur à ces actions.

Voir les politiques IAM basées sur les ressources dans Lambda

Lambda prend en charge les stratégies d'autorisations basées sur une ressource pour les fonctions et les couches Lambda. Vous pouvez utiliser des politiques basées sur les ressources pour accorder l'accès à d'autres comptes, organisations ou services AWS. Les stratégies basées sur les ressources s'appliquent à une seule fonction, version ou version de couche ou à un seul alias.

Console

Pour afficher la stratégie basée sur les ressources d'une fonction

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Sélectionnez Configuration (Configuration), puis Permissions (Autorisations).
4. Faites défiler la page vers le bas, jusqu'à Resource-based policy (Stratégie basée sur les ressources), puis choisissez View policy document (Afficher le document de stratégie). La politique basée sur les ressources indique les autorisations appliquées lorsqu'un autre compte ou AWS service tente d'accéder à la fonction. L'exemple suivant montre une

déclaration qui autorise Amazon S3 à invoquer une fonction nommée `my-function` pour un compartiment nommé `amzn-s3-demo-bucket` dans le compte `123456789012`.

Example politique basée sur les ressources

```
{
    "Version":"2012-10-17",
    "Id": "default",
    "Statement": [
        {
            "Sid": "lambda-allow-s3-my-function",
            "Effect": "Allow",
            "Principal": {
              "Service": "s3.amazonaws.com"
            },
            "Action": "lambda:InvokeFunction",
            "Resource":  "arn:aws:lambda:us-east-2:123456789012:function:my-
function",
            "Condition": {
              "StringEquals": {
                "AWS:SourceAccount": "123456789012"
              },
              "ArnLike": {
                "AWS:SourceArn": "arn:aws:s3:::amzn-s3-demo-bucket"
              }
            }
        }
     ]
}
```

AWS CLI

Pour afficher une stratégie basée sur les ressources d'une fonction, utilisez la commande `get-policy`.

```
aws lambda get-policy \
  --function-name my-function \
  --output text
```

Vous devriez voir la sortie suivante:

```
{"Version":"2012-10-17",       "Id":"default","Statement":
[{"Sid":"sns","Effect":"Allow","Principal":
{"Service":"s3.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lambd
east-2:123456789012:function:my-function","Condition":{"ArnLike":
{"AWS:SourceArn":"arn:aws:sns:us-east-2:123456789012:lambda*"}}}]}
```

Pour les versions et les alias, ajoutez le numéro de version ou l'alias au nom de la fonction.

```
aws lambda get-policy --function-name my-function:PROD
```

Pour supprimer les autorisations de votre fonction, utilisez `remove-permission`.

```
aws lambda remove-permission \
  --function-name example \
  --statement-id sns
```

Utilisez la commande `get-layer-version-policy` pour afficher les autorisations sur une couche.

```
aws lambda get-layer-version-policy \
  --layer-name my-layer \
  --version-number 3 \
  --output text
```

Vous devriez voir la sortie suivante:

```
b0cd9796-d4eb-4564-939f-de7fe0b42236    {"Sid":"engineering-
org","Effect":"Allow","Principal":"*","Action":"lambda:GetLayerVersion","Resource":"arn:aws:
west-2:123456789012:layer:my-layer:3","Condition":{"StringEquals":
{"aws:PrincipalOrgID":"o-t194hfs8cz"}}}"
```

Utilisez `remove-layer-version-permission` pour supprimer des déclarations de la stratégie.

```
aws lambda remove-layer-version-permission --layer-name my-layer --version-number 3
 --statement-id engineering-org
```

Actions d'API prises en charge

Les actions de l'API Lambda suivantes prennent en charge les politiques basées sur les ressources :

- CreateAlias
- DeleteAlias
- DeleteFunction
- DeleteFunctionConcurrency
- DeleteFunctionEventInvokeConfig
- DeleteProvisionedConcurrencyConfig
- GetAlias
- GetFunction
- GetFunctionConcurrency
- GetFunctionConfiguration
- GetFunctionEventInvokeConfig
- GetPolicy
- GetProvisionedConcurrencyConfig
- Invoke
- InvokeFunctionUrl(autorisation uniquement)
- ListAliases
- ListFunctionEventInvokeConfigs
- ListProvisionedConcurrencyConfigs
- ListTags
- ListVersionsByFunction
- PublishVersion
- PutFunctionConcurrency
- PutFunctionEventInvokeConfig
- PutProvisionedConcurrencyConfig
- TagResource
- UntagResource
- UpdateAlias
- UpdateFunctionCode

- UpdateFunctionEventInvokeConfig

Accorder l'accès à la fonction Lambda à Services AWS

Lorsque vous utilisez un AWS service pour appeler votre fonction, vous accordez l'autorisation dans une déclaration relative à une politique basée sur les ressources. Vous pouvez appliquer la déclaration à l'ensemble de la fonction, ou limiter la déclaration à une seule version ou un seul alias.

> **Note**
>
> Lorsque vous ajoutez un déclencheur à la fonction à l'aide de la console Lambda, celle-ci met à jour la stratégie basée sur une ressource de la fonction afin d'autoriser le service à l'invoquer. Pour accorder des autorisations à d'autres comptes ou services non disponibles dans la console Lambda, vous pouvez utiliser l' AWS CLI.

Ajoutez une déclaration avec la commande add-permission. La déclaration de stratégie basée sur les ressources la plus simple autorise un service à invoquer une fonction. La commande ci-après accorde à Amazon Simple Notification Service l'autorisation d'invoquer une fonction nommée `my-function`.

```
aws lambda add-permission \
  --function-name my-function \
  --action lambda:InvokeFunction \
  --statement-id sns \
  --principal sns.amazonaws.com \
  --output text
```

Vous devriez voir la sortie suivante :

```
{"Sid":"sns","Effect":"Allow","Principal":
{"Service":"sns.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lambda:us-
east-2:123456789012:function:my-function"}
```

Amazon SNS peut ainsi appeler l'action d'invocation de l'API pour la fonction, mais ne restreint pas la rubrique Amazon SNS qui déclenche l'invocation. Afin de vous assurer que votre fonction est invoquée uniquement par une ressource spécifique, spécifiez l'ARN (Amazon Resource Name) de la ressource avec l'option `source-arn`. La commande ci-après autorise uniquement Amazon SNS à invoquer la fonction pour des abonnements à une rubrique nommée `my-topic`.

```
aws lambda add-permission \
  --function-name my-function \
  --action lambda:InvokeFunction \
  --statement-id sns-my-topic \
  --principal sns.amazonaws.com \
  --source-arn arn:aws:sns:us-east-2:123456789012:my-topic
```

Certains services peuvent invoquer des fonctions dans d'autres comptes. Cela ne pose pas de problème si vous spécifiez un ARN source qui contient votre ID de compte. Pour Amazon S3, cependant, la source est un compartiment dont l'ARN ne contient pas d'ID de compte. Il est possible que vous puissiez supprimer le compartiment et qu'un autre compte crée un compartiment du même nom. Utilisez l'option `source-account` avec votre ID de compte pour vous assurer que seules les ressources de votre compte peuvent invoquer la fonction.

```
aws lambda add-permission \
  --function-name my-function \
  --action lambda:InvokeFunction \
  --statement-id s3-account \
  --principal s3.amazonaws.com \
  --source-arn arn:aws:s3:::amzn-s3-demo-bucket \
  --source-account 123456789012
```

Octroi de l'accès à la fonction à une organisation

Pour accorder des autorisations à une organisation dans AWS Organizations, spécifiez l'ID de l'organisation comme `principal-org-id`. La commande ass-permission suivante accorde un accès d'invocation à tous les utilisateurs de l'organisation `o-a1b2c3d4e5f`.

```
aws lambda add-permission \
  --function-name example \
  --statement-id PrincipalOrgIDExample \
  --action lambda:InvokeFunction \
  --principal * \
  --principal-org-id o-a1b2c3d4e5f
```

> **Note**
>
> Dans cette commande, `Principal` est `*`. Cela signifie que tous les utilisateurs de l'organisation `o-a1b2c3d4e5f` obtiennent des autorisations d'invocation de fonction. Si vous

spécifiez un rôle Compte AWS ou comme étant le`Principal`, seul ce principal obtient les autorisations d'invocation des fonctions, mais uniquement s'il fait également partie de l'o-a1b2c3d4e5forganisation.

Cette commande crée une politique basée sur les ressources qui ressemble à ce qui suit :

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "PrincipalOrgIDExample",
            "Effect": "Allow",
            "Principal": "*",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-east-2:123456789012:function:example",
            "Condition": {
                "StringEquals": {
                    "aws:PrincipalOrgID": "o-a1b2c3d4e5f"
                }
            }
        }
    ]
}
```

Pour plus d'informations, consultez aws : PrincipalOrg ID dans le guide de l'utilisateur d'IAM.

Octroi de l’accès à la fonction Lambda à d’autres comptes

Pour partager une fonction avec une autre Compte AWS, ajoutez une déclaration d'autorisations entre comptes à la politique basée sur les ressources de la fonction. Exécutez la commande add-permission et spécifiez l’ID du compte en tant que `principal`. L’exemple suivant accorde au compte `111122223333` l’autorisation d’invoquer la fonction `my-function` avec l’alias `prod`.

```
aws lambda add-permission \
  --function-name my-function:prod \
  --statement-id xaccount \
```

```
  --action lambda:InvokeFunction \
  --principal 111122223333 \
  --output text
```

Vous devriez voir la sortie suivante:

```
{"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:l
east-1:123456789012:function:my-function"}
```

La stratégie basée sur les ressources accorde à l'autre compte l'autorisation d'accéder à la fonction, mais n'autorise pas les utilisateurs de ce compte à dépasser leurs autorisations. Les utilisateurs de l'autre compte doivent disposer des autorisations utilisateur correspondantes pour utiliser l'API Lambda.

Afin de limiter l'accès à un utilisateur ou à un rôle dans un autre compte, spécifiez l'ARN complet de l'identité en tant que principal. Par exemple, `arn:aws:iam::123456789012:user/developer`.

L'alias limite la version pouvant être invoquée par l'autre compte. L'autre compte doit alors inclure l'alias dans l'ARN de la fonction.

```
aws lambda invoke \
  --function-name arn:aws:lambda:us-east-2:123456789012:function:my-function:prod out
```

Vous devriez voir la sortie suivante:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "1"
}
```

Le propriétaire de la fonction peut alors mettre à jour l'alias afin qu'il pointe vers une nouvelle version sans que l'appelant ait besoin de modifier la façon dont il invoque votre fonction. Cela permet de s'assurer que l'autre compte n'a pas besoin de changer son code pour utiliser la nouvelle version et qu'il est uniquement autorisé à invoquer la version de la fonction associée à l'alias.

Vous pouvez accorder l'accès entre comptes pour n'importe quelle action d'API qui s'exécute sur une fonction existante. Par exemple, vous pouvez accorder l'accès à `lambda:ListAliases` afin qu'un compte puisse obtenir une liste des alias ou à `lambda:GetFunction` pour qu'il puisse

télécharger le code de votre fonction. Ajoutez chaque autorisation séparément ou utilisez `lambda:*` pour accorder l'accès à toutes les actions pour la fonction spécifiée.

Pour accorder à d'autres comptes l'autorisation pour plusieurs fonctions ou pour des actions qui ne s'exécutent pas sur une fonction, nous vous conseillons d'utiliser des rôles IAM.

Octroi de l'accès de la couche Lambda à d'autres comptes

Pour partager une couche avec une autre Compte AWS, ajoutez une déclaration d'autorisations entre comptes à la politique basée sur les ressources de la couche. Exécutez la add-layer-version-permissioncommande et spécifiez l'ID du compte sous la forme`principal`. Dans chaque instruction, vous pouvez accorder une autorisation à un compte unique, à tous les comptes ou à une organisation dans AWS Organizations.

L'exemple suivant autorise le compte 111122223333 à accéder à la version 2 de la couche `bash-runtime`.

```
aws lambda add-layer-version-permission \
  --layer-name bash-runtime \
  --version-number 2 \
  --statement-id xaccount \
  --action lambda:GetLayerVersion \
  --principal 111122223333 \
  --output text
```

Vous devez voir des résultats similaires à ce qui suit :

```
{"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:GetLayerVersion","Resource":"arn:aws:
east-1:123456789012:layer:bash-runtime:2"}
```

Les autorisations ne s'appliquent qu'à une seule version de couche. Répétez le processus chaque fois que vous créez une nouvelle version de la couche.

Pour accorder l'autorisation à tous les comptes d'une organisation AWS Organizations, utilisez l'option `organization-id`. L'exemple suivant accorde à tous les comptes d'une organisation `o-t194hfs8cz` l'autorisation d'utiliser la version 3 de `my-layer`.

```
aws lambda add-layer-version-permission \
  --layer-name my-layer \
```

```
  --version-number 3 \
  --statement-id engineering-org \
  --principal '*' \
  --action lambda:GetLayerVersion \
  --organization-id o-t194hfs8cz \
  --output text
```

Vous devriez voir la sortie suivante :

```
{"Sid":"engineering-
org","Effect":"Allow","Principal":"*","Action":"lambda:GetLayerVersion","Resource":"arn:aws:lam
east-2:123456789012:layer:my-layer:3","Condition":{"StringEquals":
{"aws:PrincipalOrgID":"o-t194hfs8cz"}}}"
```

Pour octroyer l'autorisation à plusieurs comptes ou organisations, vous devez ajouter plusieurs instructions.

Utilisation du contrôle d'accès basé sur les attributs dans Lambda

Avec le Contrôle d'accès par attributs (ABAC), vous pouvez utiliser des balises pour contrôler l'accès à vos fonctions Lambda. Vous pouvez associer des balises à certaines ressources Lambda, à certaines demandes d'API ou au principal Gestion des identités et des accès AWS (IAM) émetteur de la demande. Pour plus d'informations sur la manière dont l'accès basé sur les attributs est AWS accordé, consultez la section Contrôle de l'accès aux AWS ressources à l'aide de balises dans le guide de l'utilisateur IAM.

Vous pouvez utiliser ABAC pourAccorder le moindre privilège sans spécifier ni un nom ni un modèle ARN Amazon Resource Name (ARN) dans la stratégie IAM. Au lieu de cela, vous pouvez spécifier une balise dans le champ élément de condition d'une stratégie IAM pour le contrôle de l'accès. La mise à l'échelle est plus facile avec ABAC, car vous n'avez pas à mettre à jour vos politiques IAM lorsque vous créez de nouvelles ressources. Ajoutez plutôt des balises aux nouvelles ressources pour contrôler l'accès.

Dans Lambda, les balises sont utilisées sur les ressources suivantes :

- Fonctions : pour plus d'informations sur les fonctions de balisage, voir the section called "Étiquettes".
- Configurations de signature de code : pour plus d'informations sur le balisage des configurations de signature de code, voir the section called "Balises de configuration de signature de code".

- Mappage des sources d'événements : pour plus d'informations sur le balisage du mappage des sources d'événements, voir the section called "Balises de mappage des sources d'événements".

Les balises ne sont pas prises en charge pour les couches.

Vous pouvez utiliser les clés de condition suivantes pour écrire des règles de politique IAM basées sur les balises :

- aws : ResourceTag /tag-key : contrôlez l'accès en fonction des balises associées à une ressource Lambda.
- aws : RequestTag /tag-key : exige la présence de balises dans une demande, par exemple lors de la création d'une nouvelle fonction.
- aws : PrincipalTag /tag-key : contrôlez ce que le principal IAM (la personne qui fait la demande) est autorisé à faire en fonction des balises associées à son utilisateur ou à son rôle IAM.
- aws : TagKeys : Contrôle si des clés de balise spécifiques peuvent être utilisées dans une requête.

Vous ne pouvez définir des conditions que pour les actions qui les prennent en charge. Pour obtenir une liste des conditions prises en charge par chaque action Lambda, consultez la rubrique Actions, ressources et clés de condition pour AWS Lambda dans la Référence de l'autorisation de service. Pour la prise en charge d'aws : ResourceTag /tag-key, reportez-vous à « Types de ressources définis par ». AWS Lambda Pour aws : RequestTag /tag-key et aws : TagKeys support, reportez-vous à « Actions définies par ». AWS Lambda

Rubriques

- Sécurisation de vos fonctions par balise

Sécurisation de vos fonctions par balise

Les étapes suivantes illustrent une façon de configurer des autorisations pour les fonctions à l'aide d'ABAC. Dans cet exemple de scénario, vous allez créer quatre stratégies d'autorisations IAM. Vous allez ensuite associer ces stratégies à un nouveau rôle IAM. Enfin, vous allez créer un utilisateur IAM et lui donner l'autorisation d'assumer le nouveau rôle.

Rubriques

- Conditions préalables
- Étape 1 : Exiger des balises pour les nouvelles fonctions

- Étape 2 : Autoriser les actions en fonction des balises attachées à une fonction Lambda et à un principal IAM
- Étape 3 : Accorder des permissions de liste
- Étape 4 : accorder des autorisations IAM
- Étape 5 : Création du rôle IAM
- Étape 6 : Création d'un utilisateur IAM
- Étape 7 : Tester les autorisations
- Étape 8 : Nettoyer vos ressources

Conditions préalables

Assurez-vous que vous disposez d'un rôle d'exécution Lambda. Vous utiliserez ce rôle lorsque vous accorderez des autorisations IAM et lorsque vous créerez une fonction Lambda.

Étape 1 : Exiger des balises pour les nouvelles fonctions

Lorsque vous utilisez ABAC avec Lambda, il est recommandé d'exiger que toutes les fonctions comportent des balises. Cela permet de garantir que vos politiques d'autorisation ABAC fonctionnent comme prévu.

Créez une stratégie IAM similaire à l'exemple suivant : Cette politique utilise les clés de TagKeys condition aws : RequestTag /tag-key, aws : ResourceTag /tag-key et aws : pour exiger que les nouvelles fonctions et le principal IAM qui les crée possèdent tous deux la balise. `project` Le `ForAllValues` modificateur garantit que `project` est la seule balise autorisée. Si vous n'incluez pas`ForAllValues`, les utilisateurs peuvent ajouter d'autres balises à la fonction tant qu'ils transmettent également `project`.

Example — Exiger des balises sur les nouvelles fonctions

JSON

```
{
    "Version":"2012-10-17",
    "Statement": {
      "Effect": "Allow",
      "Action": [
        "lambda:CreateFunction",
        "lambda:TagResource"
```

```
      ],
      "Resource": "arn:aws:lambda:*:*:function:*",
      "Condition": {
        "StringEquals": {
          "aws:RequestTag/project": "${aws:PrincipalTag/project}",
          "aws:ResourceTag/project": "${aws:PrincipalTag/project}"
        },
        "ForAllValues:StringEquals": {
          "aws:TagKeys": "project"
        }
      }
    }
  }
```

Étape 2 : Autoriser les actions en fonction des balises attachées à une fonction Lambda et à un principal IAM

Créez une deuxième politique IAM à l'aide de la clé de condition aws : ResourceTag /tag-key pour exiger que la balise du principal corresponde à la balise attachée à la fonction. L'exemple de stratégie suivant autorise les mandataires avec la balise `project` pour appeler des fonctions avec la balise `project`. Si une fonction possède d'autres balises, l'action est refusée.

Example — Exiger des balises correspondantes sur la fonction et le principal IAM

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
          "lambda:InvokeFunction",
          "lambda:GetFunction"
        ],
        "Resource": "arn:aws:lambda:*:*:function:*",
        "Condition": {
          "StringEquals": {
            "aws:ResourceTag/project": "${aws:PrincipalTag/project}"
          }
        }
```

```
      }
    ]
  }
```

Étape 3 : Accorder des permissions de liste

Créez une stratégie qui permet au principal de répertorier les fonctions Lambda et les rôles IAM. Cela permet au principal de voir toutes les fonctions Lambda et les rôles IAM sur la console et lors de l'appel des actions d'API.

Example — Accordez des autorisations de liste Lambda et IAM

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
      {
        "Sid": "AllResourcesLambdaNoTags",
        "Effect": "Allow",
        "Action": [
          "lambda:GetAccountSettings",
          "lambda:ListFunctions",
          "iam:ListRoles"
        ],
        "Resource": "*"
      }
    ]
  }
```

Étape 4 : accorder des autorisations IAM

Créez une politique qui autorise iam : PassRole. Cette autorisation est requise lorsque vous attribuez un rôle d'exécution à une fonction. Dans l'exemple de stratégie suivant, remplacez l'exemple d'ARN par l'ARN de votre rôle d'exécution Lambda.

> **Note**
>
> N'utilisez pas la clé de condition `ResourceTag` dans une politique avec l'action `iam:PassRole`. Vous ne pouvez pas utiliser la balise sur un rôle IAM pour contrôler l'accès

aux personnes qui peuvent transmettre ce rôle. Pour plus d'informations sur les autorisations requises pour transmettre un rôle à un service, consultez la section Octroi à un utilisateur des autorisations pour transmettre un rôle à un AWS service.

Example — Accordez l'autorisation de transmettre le rôle d'exécution

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
      {
        "Sid": "VisualEditor0",
        "Effect": "Allow",
        "Action": [
          "iam:PassRole"
        ],
        "Resource": "arn:aws:iam::111122223333:role/lambda-ex"
      }
    ]
  }
```

Étape 5 : Création du rôle IAM

C'est une bonne pratique d'utilisation des rôles pour déléguer des autorisations. Créer un rôle IAM appelé `abac-project-role` :

- Sur Étape 1 : Sélection d'une entité de confiance : Choisissez :AWS compte, puis Ce compte.
- Sur Étape 2 : Ajouter des autorisations : Joignez les quatre stratégies IAM que vous avez créées au cours des étapes précédentes.
- Sur Étape 3 : Nommer, vérifier et créer : Choisissez :Ajouter une balise. Pour Key (Clé), saisissez `project`. Ne saisissez pas Valeur.

Étape 6 : Création d'un utilisateur IAM

Création d'un utilisateur IAM appelé`abac-test-user`. Sur la page Définir des autorisations, choisissez Attacher directement les stratégies existantes, puis choisissez Créer une stratégie. Entrez

la définition de politique suivante. Remplacez *111122223333* par votre ID de compte AWS. Cette stratégie permet à `abac-test-user` d'assumer `abac-project-role`.

Example — Autoriser l'utilisateur IAM à assumer le rôle ABAC

JSON

```
{
    "Version":"2012-10-17",
    "Statement": {
      "Effect": "Allow",
      "Action": "sts:AssumeRole",
      "Resource": "arn:aws:iam::111122223333:role/abac-project-role"
    }
  }
```

Étape 7 : Tester les autorisations

1. Connectez-vous à la AWS console en tant que`abac-test-user`. Pour plus d'informations, connectez-vous en tant qu'utilisateur IAM.
2. Basculez vers le rôle `abac-project-role`. Pour plus d'informations, consultez Changement de rôles (console).
3. Création d'une fonction Lambda.

   - Sous Autorisations Choisissez Modifier le rôle d'exécution par défaut, puis dans le champ Rôle d'exécution, choisissez Utiliser un rôle existant. Choisissez le même rôle d'exécution que celui que vous avez utilisé dansÉtape 4 : accorder des autorisations IAM.
   - Sous Paramètres avancés, choisissez Activer les balises, puis Ajouter un nouveau tag. Pour Key (Clé), saisissez `project`. Ne saisissez pas Valeur.
4. Test de la fonction.
5. Créez une deuxième fonction Lambda et ajoutez une balise différente, telle que`environment`. Cette opération devrait échouer car la stratégie ABAC que vous avez créée dans Étape 1 : Exiger des balises pour les nouvelles fonctions autorise uniquement le principal à créer des fonctions avec la balise `project`.
6. Créez une troisième fonction sans balises. Cette opération devrait échouer car la stratégie ABAC que vous avez créée dansÉtape 1 : Exiger des balises pour les nouvelles fonctionsne permet pas au mandant de créer des fonctions sans balises.

Cette stratégie d'autorisation vous permet de contrôler l'accès sans créer de nouvelles politiques pour chaque nouvel utilisateur. Pour accorder l'accès à de nouveaux utilisateurs, il suffit de leur donner l'autorisation d'assumer le rôle qui correspond au projet qui leur a été attribué.

Étape 8 : Nettoyer vos ressources

Suppression du rôle IAM

1. Ouvrez la page Roles (Rôles) de la console IAM.
2. Sélectionnez le rôle que vous avez créé à l'étape 5.
3. Sélectionnez Delete (Supprimer).
4. Pour confirmer la suppression, saisissez le nom du rôle dans la zone de saisie de texte.
5. Sélectionnez Delete (Supprimer).

Pour supprimer l'utilisateur IAM

1. Ouvrez la page Utilisateurs de la console IAM.
2. Sélectionnez l'utilisateur IAM que vous avez créé à l'étape 6.
3. Sélectionnez Delete (Supprimer).
4. Pour confirmer la suppression, saisissez le nom de l'utilisateur dans la zone de saisie de texte.
5. Choisissez Delete user (Supprimer l'utilisateur).

Pour supprimer la fonction Lambda

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction que vous avez créée.
3. Sélectionnez Actions, Supprimer.
4. Saisissez **`confirm`** dans la zone de saisie de texte et choisissez Delete (Supprimer).

Optimisation des sections relatives aux stratégies de Ressources et Conditions

Vous pouvez restreindre la portée des autorisations d'un utilisateur en spécifiant des ressources et des conditions dans une stratégie IAM Gestion des identités et des accès AWS . Chaque action

d'une politique prend en charge une combinaison de types de ressources et de conditions qui varie en fonction du comportement de l'action.

Chaque déclaration de stratégie IAM accorde une autorisation pour une action effectuée sur une ressource. Lorsque l'action n'agit pas sur une ressource nommée, ou lorsque vous accordez l'autorisation d'effectuer l'action sur toutes les ressources, la valeur de la ressource de la stratégie est un caractère générique (*). Pour la plupart des actions, vous pouvez limiter les ressources qu'un utilisateur peut modifier en spécifiant l'Amazon Resource Name (ARN) d'une ressource ou un modèle ARN qui correspond à plusieurs ressources.

Par type de ressource, la conception générale pour restreindre la portée d'une action est la suivante :

- Fonctions : les actions sur une fonction peuvent être limitées à une fonction spécifique selon l'ARN de la fonction, de la version ou de l'alias.
- Mappage des sources d'événements : les actions peuvent être limitées à des ressources de mappage des sources d'événements spécifiques selon l'ARN. Le mappage des sources d'événements est toujours associé à une fonction. Vous pouvez également utiliser la condition `lambda:FunctionArn` pour restreindre les actions par fonction associée.
- Couches : les actions liées à l'utilisation et aux autorisations relatives aux couches agissent sur les versions de couche.
- Configuration de signature de code : les actions peuvent être limitées à des ressources de configuration de signature de code spécifiques selon l'ARN.
- Balises : utilisez les conditions de balise standard. Pour de plus amples informations, veuillez consulter the section called "Contrôle d'accès basé sur les attributs".

Pour limiter les autorisations par ressource, spécifiez la ressource par son ARN.

Format ARN de ressource Lambda

- Fonction – `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:function:`*`my-function`*
- Version de la fonction – `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:function:`*`my-function`*`:`*`1`*
- Alias de la fonction – `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:function:`*`my-function`*`:`*`TEST`*
- Mappage de source d'événement – `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:event-source-mapping:`*`fa123456-14a1-4fd2-9fec-83de64ad683de6d47`*

- Couche – `arn:aws:lambda:us-west-2:123456789012:layer:my-layer`
- Version de la couche – `arn:aws:lambda:us-west-2:123456789012:layer:my-layer:1`
- Configuration de signature de code – `arn:aws:lambda:us-west-2:123456789012:code-signing-config:my-csc`

Par exemple, la politique suivante permet Compte AWS `123456789012` à un utilisateur d'appeler une fonction nommée `my-function` dans la AWS région USA Ouest (Oregon).

Example appeler une stratégie de fonction

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "Invoke",
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:my-function"
        }
    ]
}
```

Il s'agit d'un cas particulier, où l'identificateur de l'action (`lambda:InvokeFunction`) diffère de l'opération d'API (Appeler). Pour les autres actions, l'identificateur de l'action est le nom de l'opération avec le préfixe `lambda:`.

Sections

- Comprendre la section Condition relative aux stratégies
- Fonctions de référencement dans la section Ressources relative aux stratégies
- Actions IAM et comportements de fonctions pris en charge

Comprendre la section Condition relative aux stratégies

Les conditions sont facultatives dans la stratégie, et appliquent une logique supplémentaire pour déterminer si une action est autorisée. Outre les conditions courantes prises en charge par toutes les actions, Lambda définit les types de condition que vous pouvez utiliser pour restreindre les valeurs des paramètres supplémentaires sur certaines actions.

Par exemple, la condition `lambda:Principal` permet de restreindre le service ou le compte auquel un utilisateur peut accorder l'accès d'invocation selon la stratégie basée sur les ressources d'une fonction. La stratégie suivante permet à un utilisateur d'accorder une autorisation à des rubriques Amazon Simple Notification Service (Amazon SNS) d'appeler une fonction nommée `test`.

Example gestion des autorisations d'une stratégie de fonction

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ManageFunctionPolicy",
            "Effect": "Allow",
            "Action": [
                "lambda:AddPermission",
                "lambda:RemovePermission"
            ],
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:test:*",
            "Condition": {
                "StringEquals": {
                    "lambda:Principal": "sns.amazonaws.com"
                }
            }
        }
    ]
}
```

La condition nécessite que le principal soit Amazon SNS et pas un autre service ou compte. Le modèle de ressource exige que le nom de la fonction soit `test` et inclue un numéro de version ou un alias. Par exemple, `test:v1`.

Pour plus d'informations sur les ressources et les conditions de Lambda et d'autres AWS services, consultez la section Actions, ressources et clés de condition pour les AWS services dans la référence d'autorisation de service.

Fonctions de référencement dans la section Ressources relative aux stratégies

Vous référencez une fonction Lambda dans une déclaration de stratégie à l'aide d'un nom Amazon Resource Name (ARN). Le format de l'ARN d'une fonction dépend du fait que vous référenciez la fonction entière, la version d'une fonction ou un alias.

Lors des appels d'API Lambda, les utilisateurs peuvent spécifier une version ou un alias en transmettant un ARN de version ou un ARN d'alias dans le GetFunction`FunctionName`paramètre, ou en définissant une valeur dans le GetFunction`Qualifier`paramètre. Lambda prend des décisions d'autorisation en comparant l'élément de ressource dans la stratégie IAM au `FunctionName` et au `Qualifier` transmis dans les appels d'API. En cas d'incompatibilité, Lambda refuse la demande.

Que vous autorisiez ou refusiez une action sur votre fonction, vous devez utiliser les types ARN de fonction appropriés dans votre déclaration de stratégie pour obtenir les résultats attendus. Par exemple, si votre stratégie fait référence à l'ARN non qualifié, Lambda accepte les demandes faisant référence à l'ARN non qualifié, mais refuse les demandes faisant référence à un ARN qualifié.

> **Note**
>
> Vous ne pouvez pas utiliser un caractère générique (*) pour établir une correspondance avec l'ID du compte. Pour plus d'informations sur la syntaxe acceptée, consultezRéférence de politique JSON IAMdans leIAM User Guide.

Example permettant l'invocation d'un ARN non qualifié

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
```

```
            "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:myFunction"
        }
    ]
}
```

Si votre stratégie fait référence à un ARN qualifié spécifique, Lambda accepte les demandes faisant référence à cet ARN, mais refuse les demandes faisant référence à l'ARN non qualifié ou à un ARN qualifié différent, par exemple,`myFunction:2`.

Example permettant l'invocation d'un ARN qualifié spécifique

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:myFunction:1"
        }
    ]
}
```

Si votre stratégie fait référence à un ARN qualifié à l'aide de `:*`, Lambda accepte tout ARN qualifié mais refuse les demandes faisant référence à l'ARN non qualifié.

Example permettant l'invocation de n'importe quel ARN qualifié

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
```

```
            "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:myFunction:*"
        }
    ]
}
```

Si votre stratégie fait référence à un ARN utilisant *, Lambda accepte tout ARN qualifié ou non qualifié.

Example permettant l'invocation de tout ARN qualifié ou non qualifié

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:myFunction*"
        }
    ]
}
```

Actions IAM et comportements de fonctions pris en charge

Les actions définissent ce qui peut être autorisé par le biais des politiques IAM. Pour obtenir une liste des actions prises en charge par Lambda, consultez la rubrique Actions, ressources et clés de condition pour AWS Lambda dans la Référence de l'autorisation de service. Dans la plupart des cas, lorsqu'une action IAM autorise une action d'API Lambda, le nom de l'action IAM est identique à celui de l'action d'API Lambda, à l'exception des actions suivantes :

Action d'API	Action IAM
Invoke	`lambda:InvokeFunction`
GetLayerVersion	`lambda:GetLayerVersion`

Action d'API	Action IAM
GetLayerVersionByArn	

Outre les ressources et les conditions définies dans la référence d'autorisation de service, Lambda prend en charge les ressources et conditions suivantes pour certaines actions. Nombre d'entre elles sont liées aux fonctions de référencement dans la section des ressources relative aux politiques. Les actions sur une fonction peuvent être limitées à une fonction spécifique selon l'ARN de la fonction, de la version ou de l'alias, tel que décrit dans le tableau suivant.

Action	Ressource	Condition
AddPermission RemovePermission Invoquer (Autorisation : `lambda:InvokeFunction`)	Version de la fonction Alias de la fonction	N/A
UpdateFunctionConfiguration	N/A	`lambda:CodeSigningConfigArn`
CreateFunctionUrlConfig DeleteFunctionUrlConfig GetFunctionUrlConfig UpdateFunctionUrlConfig	Alias de la fonction	N/A

Sécurité dans AWS Lambda

La sécurité du cloud AWS est la priorité absolue. En tant que AWS client, vous bénéficiez d'un centre de données et d'une architecture réseau conçus pour répondre aux exigences des entreprises les plus sensibles en matière de sécurité.

La sécurité est une responsabilité partagée entre vous AWS et vous. Le modèle de responsabilité partagée décrit cette notion par les termes sécurité du cloud et sécurité dans le cloud :

- Sécurité du cloud : AWS est responsable de la protection de l'infrastructure qui fonctionne Services AWS dans le AWS cloud. AWS vous fournit également des services que vous pouvez utiliser en toute sécurité. Des auditeurs tiers testent et vérifient régulièrement l'efficacité de notre sécurité dans le cadre des programmes de conformitéAWS. Pour en savoir plus sur les programmes de conformité qui s'appliquent à AWS Lambda, consultez Services AWS la section Portée par programme de conformité.
- Sécurité dans le cloud : votre responsabilité est déterminée par le service AWS que vous utilisez. Vous êtes également responsable d'autres facteurs, y compris la sensibilité de vos données, les exigences de votre entreprise,et la législation et la réglementation applicables.

Cette documentation vous aide à comprendre comment appliquer le modèle de responsabilité partagée lors de l'utilisation de Lambda. Les rubriques suivantes vous montrent comment configurer Lambda pour répondre à vos objectifs de sécurité et de conformité. Vous apprendrez également à en utiliser d'autres Services AWS qui vous aident à surveiller et à sécuriser vos ressources Lambda.

Pour plus d'informations sur l'application de principes de sécurité aux applications Lambda, Consultez Sécurité dans Serverless Land.

Rubriques

- Protection des données dans AWS Lambda
- Utilisation de rôles liés à un service pour Lambda
- Identity and Access Management (Gestion des identités et des accès) pour AWS Lambda
- Création d'une stratégie de gouvernance pour les couches et les fonctions Lambda
- Validation de la conformité pour AWS Lambda
- Résilience dans AWS Lambda
- Sécurité de l'infrastructure dans AWS Lambda

- Sécurisation des charges de travail avec des points de terminaison publics
- Utilisation de la signature de code pour vérifier l'intégrité du code avec Lambda

Protection des données dans AWS Lambda

Le modèle de responsabilité partagée AWS s'applique à la protection des données dans AWS Lambda. Comme décrit dans ce modèle, AWS est responsable de la protection de l'infrastructure globale sur laquelle l'ensemble du AWS Cloud s'exécute. La gestion du contrôle de votre contenu hébergé sur cette infrastructure relève de votre responsabilité. Vous êtes également responsable des tâches de configuration et de gestion de la sécurité des Services AWS que vous utilisez. Pour plus d'informations sur la confidentialité des données, consultez Questions fréquentes (FAQ) sur la confidentialité des données. Pour en savoir plus sur la protection des données en Europe, consultez le billet de blog Modèle de responsabilité partagée AWS et RGPD (Règlement général sur la protection des données) sur le Blog de sécurité AWS.

À des fins de protection des données, nous vous recommandons de protéger les informations d'identification Compte AWS et de configurer les comptes utilisateur individuels avec AWS IAM Identity Center ou Gestion des identités et des accès AWS (IAM). Ainsi, chaque utilisateur se voit attribuer uniquement les autorisations nécessaires pour exécuter ses tâches. Nous vous recommandons également de sécuriser vos données comme indiqué ci-dessous :

- Utilisez l'authentification multifactorielle (MFA) avec chaque compte.
- Utilisez les certificats SSL/TLS pour communiquer avec les ressources AWS. Nous exigeons TLS 1.2 et recommandons TLS 1.3.
- Configurez une API (Interface de programmation) et la journalisation des activités des utilisateurs avec AWS CloudTrail. Pour plus d'informations sur l'utilisation des sentiers CloudTrail pour capturer des activités AWS, consultez la section Utilisation des sentiers CloudTrail dans le Guide de l'utilisateur AWS CloudTrail.
- Utilisez des solutions de chiffrement AWS, ainsi que tous les contrôles de sécurité par défaut au sein des Services AWS.
- Utilisez des services de sécurité gérés avancés tels qu'Amazon Macie, qui contribuent à la découverte et à la sécurisation des données sensibles stockées dans Amazon S3.
- Si vous avez besoin de modules de chiffrement validés FIPS (Federal Information Processing Standard) 140-3 lorsque vous accédez à AWS via une interface de ligne de commande ou une API (interface de programmation), utilisez un point de terminaison FIPS. Pour plus d'informations sur

les points de terminaison FIPS disponibles, consultez Norme FIPS (Federal Information Processing Standard) 140-3.

Nous vous recommandons fortement de ne jamais placer d'informations confidentielles ou sensibles, telles que les adresses e-mail de vos clients, dans des balises ou des champs de texte libre tels que le champ Nom. Cela s'applique aussi lorsque vous utilisez Lambda ou d'autres Services AWS à l'aide de la console, de l'API, de l'AWS CLI ou des kits SDK AWS. Toutes les données que vous entrez dans des balises ou des champs de texte de forme libre utilisés pour les noms peuvent être utilisées à des fins de facturation ou dans les journaux de diagnostic. Si vous fournissez une adresse URL à un serveur externe, nous vous recommandons fortement de ne pas inclure d'informations d'identification dans l'adresse URL permettant de valider votre demande adressée à ce serveur.

Sections

- Chiffrement en transit
- Chiffrement des données au repos pour AWS Lambda

Chiffrement en transit

Les points de terminaison d'API Lambda ne prennent en charge que des connexions sécurisées sur HTTPS. Lorsque vous gérez des ressources Lambda avec l'AWS Management Console, le kit SDK AWS ou l'API Lambda, toutes les communications sont chiffrées à l'aide du protocole TLS (Transport Layer Security). Pour obtenir la liste complète des points de terminaison d'API, consultez Régions et points de terminaisonAWS dans Références générales AWS.

Lorsque vous connectez votre fonction à un système de fichiers, Lambda utilise le chiffrement pendant le transit pour toutes les connexions. Pour en savoir plus, consultez Chiffrement des données dans Amazon EFS que vous trouverez dans le guide de l'utilisateur Amazon Elastic File System.

Lorsque vous utilisez variables d'environnement, vous pouvez activer les assistants de chiffrement de la console afin qu'ils utilisent le chiffrement côté client pour protéger les variables d'environnement en transit. Pour de plus amples informations, consultez Sécurisation de variables d'environnement Lambda.

Chiffrement des données au repos pour AWS Lambda

Lambda fournit toujours un chiffrement au repos pour les ressources suivantes à l'aide d'une clé Clé détenue par AWS ou Clé gérée par AWS :

- Variables d'environnement
- Les fichiers que vous chargez sur Lambda, y compris les packages de déploiement et les archives de couches
- Les objets de critères de filtre de mappage des sources d'événements

Vous pouvez éventuellement configurer Lambda de manière à utiliser une clé gérée par le client afin de chiffrer vos variables d'environnement, vos packages de déploiement .zip et vos objets de critères de filtre.

Amazon CloudWatch Logs et AWS X-Ray chiffrent également les données par défaut, et peuvent être configurés de façon à utiliser une clé gérée par le client. Pour plus d'informations, consultez Chiffrement des données de journaux dans CloudWatch Logs et Protection des données dans AWS X-Ray.

Surveillance de vos clés de chiffrement pour Lambda

Lorsque vous utilisez une clé AWS KMS gérée par le client avec Lambda, vous pouvez utiliser AWS CloudTrail. Les exemples suivants sont des événements CloudTrail pour des appels `Decrypt`, `DescribeKey` et `GenerateDataKey` passés par Lambda pour accéder à des données chiffrées par votre clé gérée par le client.

Decrypt

Si vous avez utilisé une clé AWS KMS gérée par le client pour chiffrer votre objet de critères de filtre, Lambda envoie une requête `Decrypt` en votre nom lorsque vous essayez d'y accéder en texte brut (par exemple, à partir d'un appel `ListEventSourceMappings`). L'exemple d'événement suivant enregistre l'opération `Decrypt` :

```
{
    "eventVersion": "1.09",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROA123456789EXAMPLE:example",
        "arn": "arn:aws:sts::123456789012:assumed-role/role-name/example",
```

```
        "accountId": "123456789012",
        "accessKeyId": "ASIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROA123456789EXAMPLE",
                "arn": "arn:aws:iam::123456789012:role/role-name",
                "accountId": "123456789012",
                "userName": "role-name"
            },
            "attributes": {
                "creationDate": "2024-05-30T00:45:23Z",
                "mfaAuthenticated": "false"
            }
        },
        "invokedBy": "lambda.amazonaws.com"
    },
    "eventTime": "2024-05-30T01:05:46Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "Decrypt",
    "awsRegion": "eu-west-1",
    "sourceIPAddress": "lambda.amazonaws.com",
    "userAgent": "lambda.amazonaws.com",
    "requestParameters": {
        "keyId": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111",
        "encryptionContext": {
            "aws-crypto-public-key": "ABCD
+7876787678+CDEFGHIJKL/888666888999888555444111555222888333111==",
            "aws:lambda:EventSourceArn": "arn:aws:sqs:eu-west-1:123456789012:sample-
source",
            "aws:lambda:FunctionArn": "arn:aws:lambda:eu-
west-1:123456789012:function:sample-function"
        },
        "encryptionAlgorithm": "SYMMETRIC_DEFAULT"
    },
    "responseElements": null,
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEbbbbb",
    "readOnly": true,
    "resources": [
        {
            "accountId": "AWS Internal",
            "type": "AWS::KMS::Key",
```

```
            "ARN": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "123456789012",
    "eventCategory": "Management",
    "sessionCredentialFromConsole": "true"
}
```

DescribeKey

Si vous avez utilisé une clé AWS KMS gérée par le client pour chiffrer votre objet de critères de filtre, Lambda envoie une requête `DescribeKey` en votre nom lorsque vous essayez d'y accéder (par exemple, à partir d'un appel `GetEventSourceMapping`). L'exemple d'événement suivant enregistre l'opération `DescribeKey` :

```
{
    "eventVersion": "1.09",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROA123456789EXAMPLE:example",
        "arn": "arn:aws:sts::123456789012:assumed-role/role-name/example",
        "accountId": "123456789012",
        "accessKeyId": "ASIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROA123456789EXAMPLE",
                "arn": "arn:aws:iam::123456789012:role/role-name",
                "accountId": "123456789012",
                "userName": "role-name"
            },
            "attributes": {
                "creationDate": "2024-05-30T00:45:23Z",
                "mfaAuthenticated": "false"
            }
        }
    },
    "eventTime": "2024-05-30T01:09:40Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "DescribeKey",
```

```
    "awsRegion": "eu-west-1",
    "sourceIPAddress": "54.240.197.238",
    "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36
 (KHTML, like Gecko) Chrome/125.0.0.0 Safari/537.36",
    "requestParameters": {
        "keyId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
    },
    "responseElements": null,
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEbbbbb",
    "readOnly": true,
    "resources": [
        {
            "accountId": "AWS Internal",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "123456789012",
    "eventCategory": "Management",
    "tlsDetails": {
        "tlsVersion": "TLSv1.3",
        "cipherSuite": "TLS_AES_256_GCM_SHA384",
        "clientProvidedHostHeader": "kms.eu-west-1.amazonaws.com"
    },
    "sessionCredentialFromConsole": "true"
}
```

GenerateDataKey

Lorsque vous utilisez une clé AWS KMS gérée par le client pour chiffrer votre objet de critères de filtre dans un appel `CreateEventSourceMapping` ou `UpdateEventSourceMapping`, Lambda envoie une requête `GenerateDataKey` en votre nom pour générer une clé de données afin de chiffrer les critères de filtre (chiffrement d'enveloppe). L'exemple d'événement suivant enregistre l'opération `GenerateDataKey` :

```
{
    "eventVersion": "1.09",
    "userIdentity": {
        "type": "AssumedRole",
```

```
        "principalId": "AROA123456789EXAMPLE:example",
        "arn": "arn:aws:sts::123456789012:assumed-role/role-name/example",
        "accountId": "123456789012",
        "accessKeyId": "ASIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROA123456789EXAMPLE",
                "arn": "arn:aws:iam::123456789012:role/role-name",
                "accountId": "123456789012",
                "userName": "role-name"
            },
            "attributes": {
                "creationDate": "2024-05-30T00:06:07Z",
                "mfaAuthenticated": "false"
            }
        },
        "invokedBy": "lambda.amazonaws.com"
    },
    "eventTime": "2024-05-30T01:04:18Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "GenerateDataKey",
    "awsRegion": "eu-west-1",
    "sourceIPAddress": "lambda.amazonaws.com",
    "userAgent": "lambda.amazonaws.com",
    "requestParameters": {
        "numberOfBytes": 32,
        "keyId": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111",
        "encryptionContext": {
            "aws-crypto-public-key": "ABCD
+7876787678+CDEFGHIJKL/888666888999888555444111555222888333111==",
            "aws:lambda:EventSourceArn": "arn:aws:sqs:eu-west-1:123456789012:sample-
source",
            "aws:lambda:FunctionArn": "arn:aws:lambda:eu-
west-1:123456789012:function:sample-function"
        },
    },
    "responseElements": null,
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEbbbbb",
    "readOnly": true,
    "resources": [
        {
```

```
            "accountId": "AWS Internal",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "123456789012",
    "eventCategory": "Management"
}
```

Utilisation de rôles liés à un service pour Lambda

Lambda utilise des rôles liés à un Gestion des identités et des accès AWS service (IAM). Un rôle lié à un service est un type unique de rôle IAM directement lié à Lambda. Les rôles liés à un service sont prédéfinis par Lambda et incluent les autorisations dont le service a besoin pour appeler d'autres AWS services en votre nom.

Lambda définit les autorisations associées à ses rôles liés aux services, et seule Lambda peut assumer ses rôles. Les autorisations définies comprennent la politique de confiance et la politique d'autorisation. De plus, cette politique d'autorisation ne peut pas être attachée à une autre entité IAM.

Vous pouvez supprimer un rôle lié à un service uniquement après la suppression préalable de ses ressources connexes. Cela protège vos ressources Lambda car vous ne pouvez pas supprimer par inadvertance l'autorisation d'accès aux ressources.

Pour plus d'informations sur les autres services qui prennent en charge les rôles liés à un service, consultez la section AWS Services qui fonctionnent avec IAM et recherchez les services dont la valeur est Oui dans la colonne Rôles liés à un service. Sélectionnez un Oui ayant un lien pour consulter la documentation du rôle lié à un service, pour ce service.

Autorisations de rôle liées à un service pour Lambda

Lambda utilise le rôle lié au service nommé. AWSServiceRoleForLambda Le rôle lié à un service approuve les services suivants pour endosser le rôle :

- `lambda.amazonaws.com`

La politique d'autorisation de rôle nommée AWSLambda ServiceRolePolicy permet à Lambda d'effectuer les actions suivantes sur les ressources spécifiées :

- Action : `ec2:TerminateInstances` activée `arn:aws:ec2:*:*:instance/*` à la condition `ec2:ManagedResourceOperator` égale `scaler.lambda.amazonaws.com`
- Action : `ec2:DescribeInstanceStatus` et `ec2:DescribeInstances` sur `*`

Vous devez configurer les autorisations de manière à permettre à vos utilisateurs, groupes ou rôles de créer, modifier ou supprimer un rôle lié à un service. Pour plus d'informations, consultez Autorisations de rôles liés à un service dans le Guide de l'utilisateur IAM.

Pour les mises à jour des politiques gérées, consultez la section Politiques gérées par Lambda.

Création d'un rôle lié à un service pour Lambda

Vous n'avez pas besoin de créer manuellement un rôle lié à un service. Lorsque vous créez un fournisseur de capacité Lambda dans l'API AWS Management Console, le ou l' AWS API AWS CLI, Lambda crée le rôle lié au service pour vous.

Si vous supprimez ce rôle lié à un service et que vous avez ensuite besoin de le recréer, vous pouvez utiliser la même procédure pour recréer le rôle dans votre compte. Lorsque vous créez un fournisseur de capacité Lambda, Lambda crée à nouveau le rôle lié au service pour vous.

Vous pouvez également utiliser la console IAM pour créer un rôle lié à un service avec le AWSServiceRoleForLambdacas d'utilisation. Dans l'API AWS CLI ou dans l' AWS API, créez un rôle lié à un service avec le nom du `lambda.amazonaws.com` service. Pour plus d'informations, consultez Création d'un rôle lié à un service dans le Guide de l'utilisateur IAM. Si vous supprimez ce rôle lié à un service, vous pouvez utiliser ce même processus pour créer le rôle à nouveau.

Modification d'un rôle lié à un service pour Lambda

Lambda ne vous permet pas de modifier le rôle lié à un AWSService RoleForLambda service. Après avoir créé un rôle lié à un service, vous ne pouvez pas changer le nom du rôle, car plusieurs entités peuvent faire référence à ce rôle. Néanmoins, vous pouvez modifier la description du rôle à l'aide d'IAM. Pour plus d'informations, consultez Modification d'un rôle lié à un service dans le Guide de l'utilisateur IAM.

Supprimer un rôle lié à un service pour Lambda

Si vous n'avez plus besoin d'utiliser une fonctionnalité ou un service qui nécessite un rôle lié à un service, nous vous recommandons de supprimer ce rôle. De cette façon, vous n'avez aucune entité inutilisée qui n'est pas surveillée ou gérée activement. Cependant, vous devez nettoyer les ressources de votre rôle lié à un service avant de pouvoir les supprimer manuellement.

> **Note**
>
> Si le service Lambda utilise le rôle lorsque vous essayez de supprimer les ressources, la suppression risque d'échouer. Si cela se produit, patientez quelques minutes et réessayez.

Pour supprimer les ressources Lambda utilisées par AWSService RoleForLambda

1. Supprimez tous les fournisseurs de capacité Lambda de votre compte. Vous pouvez le faire à l'aide de la console Lambda, de la CLI ou de l'API.
2. Vérifiez qu'aucun fournisseur de capacité Lambda n'est présent dans votre compte avant de tenter de supprimer le rôle lié à un service.

Pour supprimer manuellement le rôle lié au service à l'aide d'IAM

Utilisez la console IAM, le AWS CLI, ou l' AWS API pour supprimer le rôle lié au AWSService RoleForLambda service. Pour plus d'informations, consultez la section Suppression d'un rôle lié à un service dans le Guide de l'utilisateur IAM.

Régions prises en charge pour les rôles liés à un service Lambda

Lambda ne prend pas en charge l'utilisation de rôles liés à un service dans toutes les régions où le service est disponible. AWSServiceRoleForLambda est pris en charge dans les régions suivantes.

Nom de la région	Identité de la région	Support dans Lambda
USA Est (Virginie du Nord)	us-east-1	Oui
USA Est (Ohio)	us-east-2	Oui
USA Ouest (Californie du Nord)	us-west-1	Oui

Nom de la région	Identité de la région	Support dans Lambda
USA Ouest (Oregon)	us-west-2	Oui
Afrique (Le Cap)	af-south-1	Non
Asie-Pacifique (Hong Kong)	ap-east-1	Oui
Asie-Pacifique (Jakarta)	ap-southeast-3	Oui
Asie-Pacifique (Bangkok)	ap-southeast-7	Oui
Asie-Pacifique (Mumbai)	ap-south-1	Oui
Asie-Pacifique (Osaka)	ap-northeast-3	Non
Asie-Pacifique (Séoul)	ap-northeast-2	Non
Asie-Pacifique (Singapour)	ap-southeast-1	Oui
Asie-Pacifique (Sydney)	ap-southeast-2	Oui
Asie-Pacifique (Tokyo)	ap-northeast-1	Oui
Canada (Centre)	ca-central-1	Non
Europe (Francfort)	eu-central-1	Oui
Europe (Irlande)	eu-west-1	Oui
Europe (Londres)	eu-west-2	Oui
Europe (Milan)	eu-south-1	Non
Europe (Paris)	eu-west-3	Non
Europe (Stockholm)	eu-north-1	Non
Moyen-Orient (Bahreïn)	me-south-1	Non
Moyen-Orient (EAU)	me-central-1	Non

Nom de la région	Identité de la région	Support dans Lambda
Amérique du Sud (São Paulo)	sa-east-1	Non
AWS GovCloud (USA Est)	us-gov-east-1	Non
AWS GovCloud (US-Ouest)	us-gov-west-1	Non

Identity and Access Management (Gestion des identités et des accès) pour AWS Lambda

Gestion des identités et des accès AWS (IAM) est un Service AWS qui aide un administrateur à contrôler en toute sécurité l'accès aux ressources AWS. Des administrateurs IAM contrôlent les personnes qui s'authentifient (sont connectées) et sont autorisées (disposent d'autorisations) à utiliser des ressources . IAM est un Service AWS que vous pouvez utiliser sans frais supplémentaires.

Rubriques

- Public ciblé
- Authentification par des identités
- Gestion des accès à l'aide de politiques
- Fonctionnement d'AWS Lambda avec IAM
- Exemples de politiques basées sur l'identité pour AWS Lambda
- AWS politiques gérées pour AWS Lambda
- Résolution des problèmes d'identité et d'accès avec AWS Lambda

Public ciblé

La façon dont vous utilisez Gestion des identités et des accès AWS (IAM) varie en fonction de votre rôle :

- Utilisateur du service : demandez des autorisations à votre administrateur si vous ne pouvez pas accéder aux fonctionnalités (voir Résolution des problèmes d'identité et d'accès avec AWS Lambda)

- Administrateur du service : déterminez l'accès des utilisateurs et soumettez les demandes d'autorisation (voir Fonctionnement d'AWS Lambda avec IAM)
- Administrateur IAM : rédigez des politiques pour gérer l'accès (voir Exemples de politiques basées sur l'identité pour AWS Lambda)

Authentification par des identités

L'authentification correspond au processus par lequel vous vous connectez à AWS avec vos informations d'identification. Vous devez vous authentifier en tant qu'Utilisateur racine d'un compte AWS, en tant qu'utilisateur IAM ou en endossant un rôle IAM.

Vous pouvez vous connecter en tant qu'identité fédérée en utilisant les informations d'identification provenant d'une source d'identité telle que AWS IAM Identity Center (IAM Identity Center), l'authentification unique ou les informations d'identification Google/Facebook. Pour plus d'informations sur la connexion, consultez Connexion à votre Compte AWS dans le Guide de l'utilisateur Connexion à AWS.

Pour l'accès par programmation, AWS fournit un kit SDK et une CLI pour signer les demandes de manière cryptographique. Pour plus d'informations, consultez Signature AWS Version 4 pour les demandes d'API dans le Guide de l'utilisateur IAM.

Utilisateur racine Compte AWS

Lorsque vous créez un Compte AWS, vous commencez avec une seule identité de connexion nommée utilisateur racine du Compte AWS, qui bénéficie d'un accès complet à tous les Services AWS et toutes les ressources du compte. Il est vivement déconseillé d'utiliser l'utilisateur racine pour vos tâches quotidiennes. Pour les tâches qui requièrent des informations d'identification de l'utilisateur racine, consultez Tâches qui requièrent les informations d'identification de l'utilisateur racine dans le Guide de l'utilisateur IAM.

Identité fédérée

Nous vous recommandons vivement d'exiger de vos utilisateurs humains qu'ils utilisent une fédération liée à un fournisseur d'identité pour accéder à Services AWS avec des informations d'identification temporaires.

Une identité fédérée est un utilisateur provenant de l'annuaire de votre entreprise, de votre fournisseur d'identité Web ou Directory Service qui y accède à Services AWS l'aide d'informations

d'identification provenant d'une source d'identité. Les identités fédérées assument des rôles qui fournissent des informations d'identification temporaires.

Pour une gestion des accès centralisée, nous vous recommandons d'utiliser AWS IAM Identity Center. Pour plus d'informations, consultez Qu'est-ce que IAM Identity Center ? dans le Guide de l'utilisateur AWS IAM Identity Center.

Utilisateurs et groupes IAM

Un utilisateur IAM est une identité qui dispose d'autorisations spécifiques pour une seule personne ou application. Nous vous recommandons d'utiliser ces informations d'identification temporaires au lieu des utilisateurs IAM avec des informations d'identification à long terme. Pour plus d'informations, consultez Exiger des utilisateurs humains qu'ils utilisent une fédération avec un fournisseur d'identité pour accéder à AWS en utilisant des informations d'identification temporaires dans le Guide de l'utilisateur IAM.

Les groupes IAM spécifient une collection d'utilisateurs IAM et permettent de gérer plus facilement les autorisations pour de grands ensembles d'utilisateurs. Pour plus d'informations, consultez la section Cas d'utilisation pour les utilisateurs IAM dans le Guide de l'utilisateur IAM.

Rôles IAM

Un rôle IAM est une identité dotée d'autorisations spécifiques qui fournit des informations d'identification temporaires. Pour endosser un rôle, passez d'un utilisateur à un rôle IAM (console) ou appelez une AWS CLI ou une opération d'API AWS. Pour plus d'informations, consultez Méthodes pour endosser un rôle dans le Guide de l'utilisateur IAM.

Les rôles IAM sont utiles pour l'accès des utilisateurs fédérés, les autorisations temporaires des utilisateurs IAM, les accès entre comptes, les accès entre services et les applications exécutées sur Amazon EC2. Pour plus d'informations, consultez Accès intercompte aux ressources dans IAM dans le Guide de l'utilisateur IAM.

Gestion des accès à l'aide de politiques

Vous contrôlez les accès dans AWS en créant des politiques et en les attachant à des identités AWS ou à des ressources. Une politique définit les autorisations lorsqu'elles sont associées à une identité ou une ressource. AWS évalue ces politiques lorsqu'un principal effectue une demande. La plupart des politiques sont stockées dans AWS en tant que documents JSON. Pour plus d'informations les documents de politique JSON, consultez Vue d'ensemble des politiques JSON dans le Guide de l'utilisateur IAM.

À l'aide de politiques, les administrateurs précisent qui a accès à quoi en définissant quel principal peut effectuer des actions sur quelles ressources et dans quelles conditions.

Par défaut, les utilisateurs et les rôles ne disposent d'aucune autorisation. Un administrateur IAM crée des politiques IAM et les ajoute aux rôles, que les utilisateurs peuvent ensuite assumer. Les politiques IAM définissent les autorisations quelle que soit la méthode que vous utilisez pour exécuter l'opération.

Politiques basées sur l'identité

Les stratégies basées sur l'identité sont des documents de stratégie d'autorisations JSON que vous attachez à une identité (utilisateur, groupe ou rôle). Ces politiques contrôlent les actions que peuvent exécuter ces identités, sur quelles ressources et dans quelles conditions. Pour découvrir comment créer une politique basée sur l'identité, consultez Définition d'autorisations IAM personnalisées avec des politiques gérées par le client dans le Guide de l'utilisateur IAM.

Les politiques basées sur l'identité peuvent être des politiques intégrées (intégrées directement dans une seule identité) ou des politiques gérées (politiques autonomes associées à plusieurs identités). Pour découvrir comment choisir entre des politiques gérées et en ligne, consultez Choix entre les politiques gérées et les politiques en ligne dans le Guide de l'utilisateur IAM.

Politiques basées sur les ressources

Les politiques basées sur les ressources sont des documents de politique JSON que vous attachez à une ressource. Les exemples incluent les politiques de confiance de rôle IAM et les politiques de compartiment Amazon S3. Dans les services qui sont compatibles avec les politiques basées sur les ressources, les administrateurs de service peuvent les utiliser pour contrôler l'accès à une ressource spécifique. Vous devez spécifier un principal dans une politique basée sur les ressources.

Les politiques basées sur les ressources sont des politiques en ligne situées dans ce service. Vous ne pouvez pas utiliser les politiques gérées AWS depuis IAM dans une politique basée sur une ressource.

Autres types de politique

AWS prend en charge des types de politiques supplémentaires qui peuvent définir le nombre maximum d'autorisations qui vous sont accordées par des types de politiques plus courants :

- Limites d'autorisations : une limite des autorisations définit le nombre maximum d'autorisations qu'une politique basée sur l'identité peut accorder à une entité IAM. Pour plus d'informations, consultez Limites d'autorisations pour des entités IAM dans le Guide de l'utilisateur IAM.

- Politiques de contrôle de service (SCP) : spécifient les autorisations maximales pour une organisation ou une unité organisationnelle dans AWS Organizations. Pour plus d'informations, veuillez consulter Politiques de contrôle de service du Guide de l'utilisateur AWS Organizations.
- Politiques de contrôle des ressources (RCP) : définissent les autorisations maximales disponibles pour les ressources de votre organisation. Pour plus d'informations, consultez Politiques de contrôle des ressources (RCP) dans le Guide de l'utilisateur AWS Organizations.
- Politiques de session : politiques avancées que vous passez en tant que paramètre lorsque vous créez par programmation une session temporaire pour un rôle ou un utilisateur fédéré. Pour plus d'informations, consultez Politiques de session dans le Guide de l'utilisateur IAM.

Plusieurs types de politique

Lorsque plusieurs types de politiques s'appliquent à la requête, les autorisations en résultant sont plus compliquées à comprendre. Pour découvrir la façon dont AWS détermine s'il convient d'autoriser une demande en présence de plusieurs types de politiques, consultez Logique d'évaluation de politiques dans le Guide de l'utilisateur IAM.

Fonctionnement d'AWS Lambda avec IAM

Avant d'utiliser IAM pour gérer l'accès à Lambda, découvrez les fonctions IAM que vous pouvez utiliser avec Lambda.

Fonctionnalité IAM	Support Lambda
Politiques basées sur l'identité	Oui
Politiques basées sur les ressources	Oui
Actions de politique	Oui
Ressources de politique	Oui
Clés de condition de politique (spécifiques au service)	Oui
ACL	Non
ABAC (identifications dans les politiques)	Partielle

Fonctionnalité IAM	Support Lambda
Informations d'identification temporaires	Oui
Transfert des sessions d'accès (FAS)	Non
Rôles de service	Oui
Rôles liés à un service	Partielle

Pour obtenir une vue d'ensemble de la façon dont Lambda et d'autres services AWS fonctionnent avec la plupart des fonctionnalités IAM, consultez Services AWS qui fonctionnent avec IAM dans le Guide de l'utilisateur IAM.

Politiques basées sur l'identité pour Lambda

Prend en charge les politiques basées sur l'identité : oui

Les politiques basées sur l'identité sont des documents de politique d'autorisations JSON que vous pouvez attacher à une identité telle qu'un utilisateur, un groupe d'utilisateurs ou un rôle IAM. Ces politiques contrôlent quel type d'actions des utilisateurs et des rôles peuvent exécuter, sur quelles ressources et dans quelles conditions. Pour découvrir comment créer une politique basée sur l'identité, consultez Définition d'autorisations IAM personnalisées avec des politiques gérées par le client dans le Guide de l'utilisateur IAM.

Avec les politiques IAM basées sur l'identité, vous pouvez spécifier des actions et ressources autorisées ou refusées, ainsi que les conditions dans lesquelles les actions sont autorisées ou refusées. Pour découvrir tous les éléments que vous utilisez dans une politique JSON, consultez Références des éléments de politique JSON IAM dans le Guide de l'utilisateur IAM.

Exemples de politiques basées sur l'identité pour Lambda

Pour voir des exemples de politiques Lambda basées sur l'identité, consultez Exemples de politiques basées sur l'identité pour AWS Lambda.

Politiques basées sur les ressources dans Lambda

Prend en charge les politiques basées sur les ressources : oui

Les politiques basées sur les ressources sont des documents de politique JSON que vous attachez à une ressource. Par exemple, les politiques de confiance de rôle IAM et les politiques de compartiment Amazon S3 sont des politiques basées sur les ressources. Dans les services qui sont compatibles avec les politiques basées sur les ressources, les administrateurs de service peuvent les utiliser pour contrôler l'accès à une ressource spécifique. Pour la ressource dans laquelle se trouve la politique, cette dernière définit quel type d'actions un principal spécifié peut effectuer sur cette ressource et dans quelles conditions. Vous devez spécifier un principal dans une politique basée sur les ressources. Les principaux peuvent inclure des comptes, des utilisateurs, des rôles, des utilisateurs fédérés ou des Services AWS.

Pour permettre un accès intercompte, vous pouvez spécifier un compte entier ou des entités IAM dans un autre compte en tant que principal dans une politique basée sur les ressources. Pour plus d'informations, consultez Accès intercompte aux ressources dans IAM dans le Guide de l'utilisateur IAM.

Vous pouvez attacher une politique basée sur les ressources à une fonction ou une couche Lambda. Cette politique désigne les principaux autorisés à effectuer des actions sur la fonction ou la couche.

Pour savoir comment attacher une politique basée sur les ressources à une fonction ou une couche, consultez Voir les politiques IAM basées sur les ressources dans Lambda.

Actions de politique pour Lambda

Prend en charge les actions de politique : oui

Les administrateurs peuvent utiliser les politiques JSON AWS pour spécifier qui a accès à quoi. C'est-à-dire, quel principal peut effectuer des actions sur quelles ressources et dans quelles conditions.

L'élément `Action` d'une politique JSON décrit les actions que vous pouvez utiliser pour autoriser ou refuser l'accès à une politique. Intégration d'actions dans une politique afin d'accorder l'autorisation d'exécuter les opérations associées.

Pour afficher la liste des actions Lambda, consultez Actions définies par AWS Lambda dans la Référence de l'autorisation de service.

Les actions de politique dans Lambda utilisent le préfixe suivant avant l'action :

```
lambda
```

Pour indiquer plusieurs actions dans une seule déclaration, séparez-les par des virgules.

```
"Action": [
      "lambda:action1",
      "lambda:action2"
         ]
```

Pour voir des exemples de politiques Lambda basées sur l'identité, consultez Exemples de politiques basées sur l'identité pour AWS Lambda.

Ressources de politique pour Lambda

Prend en charge les ressources de politique : oui

Les administrateurs peuvent utiliser les politiques JSON AWS pour spécifier qui a accès à quoi. C'est-à-dire, quel principal peut effectuer des actions sur quelles ressources et dans quelles conditions.

L'élément de politique JSON `Resource` indique le ou les objets auxquels l'action s'applique. Il est recommandé de définir une ressource à l'aide de son Amazon Resource Name (ARN). Pour les actions qui ne sont pas compatibles avec les autorisations de niveau ressource, utilisez un caractère générique (*) afin d'indiquer que l'instruction s'applique à toutes les ressources.

```
"Resource": "*"
```

Pour afficher la liste des types de ressources Lambda et leurs ARN, consultez Types de ressources définies par AWS Lambda dans la Référence de l'autorisation de service. Pour savoir grâce à quelles actions vous pouvez spécifier l'ARN de chaque ressource, consultez Actions définies par AWS Lambda.

Pour voir des exemples de politiques Lambda basées sur l'identité, consultez Exemples de politiques basées sur l'identité pour AWS Lambda.

Clés de condition de politique pour Lambda

Prend en charge les clés de condition de politique spécifiques au service : oui

Les administrateurs peuvent utiliser les politiques JSON AWS pour spécifier qui a accès à quoi. C'est-à-dire, quel principal peut effectuer des actions sur quelles ressources et dans quelles conditions.

L'élément `Condition` indique à quel moment les instructions sont exécutées en fonction de critères définis. Vous pouvez créer des expressions conditionnelles qui utilisent des opérateurs de condition, tels que les signes égal ou inférieur à, pour faire correspondre la condition de la politique aux valeurs de la demande. Pour afficher toutes les clés de condition globales AWS, consultez Clés de contexte de condition globales AWS dans le Guide de l'utilisateur IAM.

Pour afficher la liste des clés de condition Lambda, consultez Clés de condition pour AWS Lambda dans la Référence de l'autorisation de service. Pour savoir avec quelles actions et ressources vous pouvez utiliser une clé de condition, consultez Actions définies par AWS Lambda.

Pour voir des exemples de politiques Lambda basées sur l'identité, consultez Exemples de politiques basées sur l'identité pour AWS Lambda.

Listes ACL dans Lambda

Prend en charge les ACL : non

Les listes de contrôle d'accès (ACL) vérifie quels principaux (membres de compte, utilisateurs ou rôles) ont l'autorisation d'accéder à une ressource. Les listes de contrôle d'accès sont similaires aux politiques basées sur les ressources, bien qu'elles n'utilisent pas le format de document de politique JSON.

ABAC avec Lambda

Prend en charge ABAC (identifications dans les politiques) : partiellement

Le contrôle d'accès par attributs (ABAC) est une stratégie d'autorisation qui définit les autorisations en fonction des attributs nommés balise. Vous pouvez attacher des balises aux entités IAM et aux ressources AWS, puis concevoir des politiques ABAC pour autoriser les opérations lorsque la balise du principal correspond à celle de la ressource.

Pour contrôler l'accès basé sur des étiquettes, vous devez fournir les informations d'étiquette dans l'élément de condition d'une politique utilisant les clés de condition `aws:ResourceTag/`*`key-name`*, `aws:RequestTag/`*`key-name`* ou `aws:TagKeys`.

Si un service prend en charge les trois clés de condition pour tous les types de ressources, alors la valeur pour ce service est Oui. Si un service prend en charge les trois clés de condition pour certains types de ressources uniquement, la valeur est Partielle.

Pour plus d'informations sur ABAC, consultez Définition d'autorisations avec l'autorisation ABAC dans le Guide de l'utilisateur IAM. Pour accéder à un didacticiel décrivant les étapes de configuration de l'ABAC, consultez Utilisation du contrôle d'accès par attributs (ABAC) dans le Guide de l'utilisateur IAM.

Pour plus d'informations sur le balisage des ressources Lambda, consultez Utilisation du contrôle d'accès basé sur les attributs dans Lambda.

Utilisation des informations d'identification temporaires avec Lambda

Prend en charge les informations d'identification temporaires : oui

Les informations d'identification temporaires fournissent un accès à court terme aux ressources AWS et sont automatiquement créées lorsque vous utilisez la fédération ou que vous changez de rôle. AWS recommande de générer des informations d'identification temporaires dynamiquement pour utiliser des clés d'accès à long terme. Pour plus d'informations, consultez la section Informations d'identification de sécurité temporaires dans IAM et Services AWS compatibles avec IAM dans le Guide de l'utilisateur IAM.

Transmission des sessions d'accès pour Lambda

Prend en charge les sessions d'accès direct (FAS) : oui

Les sessions FAS utilisent les autorisations du principal en appelant un Service AWS, associé au Service AWS demandeur afin d'effectuer des demandes aux services en aval. Pour plus de détails sur la politique relative à la transmission de demandes FAS, consultez la section Sessions de transmission d'accès.

Rôles de service pour Lambda

Prend en charge les rôles de service : oui

Un rôle de service est un rôle IAM qu'un service endosse pour accomplir des actions en votre nom. Un administrateur IAM peut créer, modifier et supprimer un rôle de service à partir d'IAM. Pour plus d'informations, consultez Création d'un rôle pour la délégation d'autorisations à un Service AWS dans le Guide de l'utilisateur IAM.

Dans Lambda, un rôle de service est appelé rôle d'exécution.

> **Warning**
>
> La modification des autorisations d'un rôle d'exécution peut altérer la fonction Lambda.

Rôles liés à un service pour Lambda

Prend en charge les rôles liés à un service : partiellement

Un rôle lié à un service est un type de rôle de service lié à un Service AWS. Le service peut endosser le rôle afin d'effectuer une action en votre nom. Les rôles liés à un service s'affichent dans votre Compte AWS et sont détenus par le service. Un administrateur IAM peut consulter, mais ne peut pas modifier, les autorisations concernant les rôles liés à un service.

Lambda n'a pas de rôles liés au service, contrairement à Lambda@Edge. Pour plus d'informations, veuillez consulter la rubrique Rôles liés à un service pour Lambda@Edge dans le Guide du développeur Amazon CloudFront.

Pour plus d'informations sur la création ou la gestion des rôles liés à un service, consultez Services AWS qui fonctionnent avec IAM. Recherchez un service dans le tableau qui inclut un `Yes` dans la colonne Rôle lié à un service. Choisissez le lien Oui pour consulter la documentation du rôle lié à ce service.

Exemples de politiques basées sur l'identité pour AWS Lambda

Par défaut, les utilisateurs et les rôles ne sont pas autorisés à créer ou à modifier des ressources Lambda. Pour octroyer aux utilisateurs des autorisations d'effectuer des actions sur les ressources dont ils ont besoin, un administrateur IAM peut créer des politiques IAM.

Pour apprendre à créer une politique basée sur l'identité IAM à l'aide de ces exemples de documents de politique JSON, consultez Création de politiques IAM (console) dans le Guide de l'utilisateur IAM.

Pour plus de détails sur les actions et les types de ressources définis par Lambda, y compris le format des ARN pour chacun des types de ressources, veuillez consulter la rubrique Actions, ressources et clés de condition pour AWS Lambda dans la Référence de l'autorisation de service.

Rubriques

- Bonnes pratiques en matière de politiques
- Utilisation de la console Lambda
- Autorisation accordée aux utilisateurs pour afficher leurs propres autorisations

Bonnes pratiques en matière de politiques

Les stratégies basées sur l'identité déterminent si une personne peut créer, consulter ou supprimer des ressources Lambda dans votre compte. Ces actions peuvent entraîner des frais pour votre Compte AWS. Lorsque vous créez ou modifiez des politiques basées sur l'identité, suivez ces instructions et recommandations :

- Démarrez avec les politiques gérées par AWS et évoluez vers les autorisations de moindre privilège : pour commencer à accorder des autorisations à vos utilisateurs et charges de travail, utilisez les politiques gérées par AWS qui accordent des autorisations dans de nombreux cas d'utilisation courants. Elles sont disponibles dans votre Compte AWS. Nous vous recommandons de réduire encore les autorisations en définissant des politiques AWS gérées par le client qui sont propres à vos cas d'utilisation. Pour plus d'informations, consultez politiques gérées par AWS ou politiques gérées par AWS pour les activités professionnelles dans le Guide de l'utilisateur IAM.
- Accordez les autorisations de moindre privilège : lorsque vous définissez des autorisations avec des politiques IAM, accordez uniquement les autorisations nécessaires à l'exécution d'une seule tâche. Pour ce faire, vous définissez les actions qui peuvent être entreprises sur des ressources spécifiques dans des conditions spécifiques, également appelées autorisations de moindre privilège. Pour plus d'informations sur l'utilisation d'IAM pour appliquer des autorisations, consultez politiques et autorisations dans IAM dans le Guide de l'utilisateur IAM.
- Utilisez des conditions dans les politiques IAM pour restreindre davantage l'accès : vous pouvez ajouter une condition à vos politiques afin de limiter l'accès aux actions et aux ressources. Par exemple, vous pouvez écrire une condition de politique pour spécifier que toutes les demandes doivent être envoyées via SSL. Vous pouvez également utiliser des conditions pour accorder l'accès aux actions de service si elles sont utilisées via un Service AWS spécifique, comme CloudFormation. Pour plus d'informations, consultez Conditions pour éléments de politique JSON IAM dans le Guide de l'utilisateur IAM.
- Utilisez l'Analyseur d'accès IAM pour valider vos politiques IAM afin de garantir des autorisations sécurisées et fonctionnelles : l'Analyseur d'accès IAM valide les politiques nouvelles et existantes de manière à ce que les politiques IAM respectent le langage de politique IAM (JSON) et les bonnes pratiques IAM. IAM Access Analyzer fournit plus de 100 vérifications de politiques et des recommandations exploitables pour vous aider à créer des politiques sécurisées et fonctionnelles.

Pour plus d'informations, consultez Validation de politiques avec IAM Access Analyzer dans le Guide de l'utilisateur IAM.

- Exigez l'authentification multifactorielle (MFA) : si vous avez un scénario qui nécessite des utilisateurs IAM ou un utilisateur racine dans votre Compte AWS, activez l'authentification multifactorielle pour une sécurité renforcée. Pour exiger la MFA lorsque des opérations d'API sont appelées, ajoutez des conditions MFA à vos politiques. Pour plus d'informations, consultez Sécurisation de l'accès aux API avec MFA dans le Guide de l'utilisateur IAM.

Pour plus d'informations sur les bonnes pratiques dans IAM, consultez Bonnes pratiques de sécurité dans IAM dans le Guide de l'utilisateur IAM.

Utilisation de la console Lambda

Pour accéder à la console AWS Lambda, vous devez disposer d'un ensemble minimum d'autorisations. Ces autorisations doivent vous permettre de répertorier et de consulter les informations relatives aux ressources Lambda de votre Compte AWS. Si vous créez une politique basée sur l'identité qui est plus restrictive que l'ensemble minimum d'autorisations requis, la console ne fonctionnera pas comme prévu pour les entités (utilisateurs ou rôles) tributaires de cette politique.

Vous n'avez pas besoin d'accorder les autorisations minimales de console pour les utilisateurs qui effectuent des appels uniquement à AWS CLI ou à l'API AWS. Autorisez plutôt l'accès à uniquement aux actions qui correspondent à l'opération d'API qu'ils tentent d'effectuer.

Pour obtenir un exemple de stratégie qui accorde un accès minimal pour le développement de fonctions, veuillez consulter Octroi aux utilisateurs de l'accès à une fonction Lambda. En plus des API Lambda, la console Lambda utilise d'autres services pour afficher la configuration des déclencheurs et vous permettre d'en ajouter de nouveaux. Si vos utilisateurs utilisent Lambda avec d'autres services, ils ont également besoin d'accéder à ces services. Pour plus d'informations sur la configuration d'autres services avec Lambda, consultez Invoquer Lambda avec des événements provenant d'autres services AWS.

Autorisation accordée aux utilisateurs pour afficher leurs propres autorisations

Cet exemple montre comment créer une politique qui permet aux utilisateurs IAM d'afficher les politiques en ligne et gérées attachées à leur identité d'utilisateur. Cette stratégie inclut les autorisations nécessaires pour réaliser cette action sur la console ou par programmation à l'aide de l'AWS CLI ou de l'API AWS.

```
{
```

```
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS politiques gérées pour AWS Lambda

Une politique AWS gérée est une politique autonome créée et administrée par AWS. AWS les politiques gérées sont conçues pour fournir des autorisations pour de nombreux cas d'utilisation courants afin que vous puissiez commencer à attribuer des autorisations aux utilisateurs, aux groupes et aux rôles.

N'oubliez pas que les politiques AWS gérées peuvent ne pas accorder d'autorisations de moindre privilège pour vos cas d'utilisation spécifiques, car elles sont accessibles à tous les AWS clients.

Nous vous recommandons de réduire encore les autorisations en définissant des politiques gérées par le client qui sont propres à vos cas d'utilisation.

Vous ne pouvez pas modifier les autorisations définies dans les politiques AWS gérées. Si les autorisations définies dans une politique AWS gérée sont AWS mises à jour, la mise à jour affecte toutes les identités principales (utilisateurs, groupes et rôles) auxquelles la politique est attachée. AWS est le plus susceptible de mettre à jour une politique AWS gérée lorsqu'une nouvelle Service AWS est lancée ou lorsque de nouvelles opérations d'API sont disponibles pour les services existants.

Pour plus d'informations, consultez Politiques gérées par AWS dans le Guide de l'utilisateur IAM.

Rubriques

- AWS politique gérée : AWSLambda_FullAccess
- AWS politique gérée : AWSLambda_ReadOnlyAccess
- AWS politique gérée : AWSLambda BasicExecutionRole
- AWS politique gérée : AWSLambda BasicDurableExecutionRolePolicy
- AWS stratégie gérée : AWSLambda Dynamo Role DBExecution
- AWS politique gérée : AWSLambda ENIManagement Accès
- AWS politique gérée : AWSLambdaInvocation-DynamoDB
- AWS politique gérée : AWSLambda KinesisExecutionRole
- AWS politique gérée : AWSLambda MSKExecution rôle
- AWS politique gérée : AWSLambda rôle
- AWS politique gérée : AWSLambda SQSQueue ExecutionRole
- AWS politique gérée : AWSLambda VPCAccess ExecutionRole
- AWS politique gérée : AWSLambda Géré EC2 ResourceOperator
- AWS politique gérée : AWSLambda ServiceRolePolicy
- Mises à jour Lambda des politiques gérées AWS

AWS politique gérée : AWSLambda_FullAccess

Cette stratégie accorde un accès complet aux actions Lambda. Il accorde également des autorisations à d'autres AWS services utilisés pour développer et gérer les ressources Lambda.

Vous pouvez attacher la stratégie `AWSLambda_FullAccess` à vos utilisateurs, groupes et rôles.

Détails de l’autorisation

Cette politique inclut les autorisations suivantes :

- `lambda` : donne aux principaux un accès complet à Lambda.
- `cloudformation`— Permet aux directeurs de décrire les AWS CloudFormation piles et de répertorier les ressources qu'elles contiennent.
- `cloudwatch`— Permet aux principaux de répertorier les CloudWatch métriques Amazon et d'obtenir des données métriques.
- `ec2`— Permet aux principaux de décrire les groupes de sécurité, les sous-réseaux et. VPCs
- `iam` : permet aux principaux d’obtenir les stratégies, les versions des stratégies, les rôles, les stratégies de rôle, les stratégies de rôles associées et la liste des rôles. Cette stratégie permet également aux principaux de transmettre des rôles à Lambda. L’autorisation `PassRole` est utilisée lorsque vous attribuez un rôle d’exécution à une fonction. L'`CreateServiceLinkedRole`autorisation est utilisée lors de la création d'un rôle lié à un service.
- `kms`— Permet aux principaux de répertorier les alias et de décrire la clé pour le chiffrement des volumes.
- `logs` : permet aux principaux de décrire les flux de journaux, d’obtenir les événements du journal, de filtrer les événements du journal et de démarrer et arrêter dsessions Live Tail.
- `states`— Permet aux directeurs de décrire et de répertorier les machines AWS Step Functions d'état.
- `tag` : permet aux principaux d’obtenir des ressources en fonction de leurs balises.
- `xray`— Permet aux principaux d'obtenir des résumés de AWS X-Ray traces et de récupérer une liste de traces spécifiée par ID.

Pour plus d'informations sur cette politique, y compris le document de politique JSON et les versions de politique, consultez AWSLambda_FullAccessle Guide de référence des politiques AWS gérées.

AWS politique gérée : AWSLambda_ReadOnlyAccess

Cette politique accorde un accès en lecture seule aux ressources Lambda et aux autres AWS services utilisés pour développer et gérer les ressources Lambda.

Vous pouvez attacher la stratégie `AWSLambda_ReadOnlyAccess` à vos utilisateurs, groupes et rôles.

Détails de l’autorisation

Cette politique inclut les autorisations suivantes :

- `lambda` : permet aux principaux d’obtenir et de répertorier toutes les ressources.
- `cloudformation`— Permet aux directeurs de décrire et de répertorier les AWS CloudFormation piles et de répertorier les ressources qu'elles contiennent.
- `cloudwatch`— Permet aux principaux de répertorier les CloudWatch métriques Amazon et d'obtenir des données métriques.
- `ec2`— Permet aux principaux de décrire les groupes de sécurité, les sous-réseaux et. VPCs
- `iam` : permet aux principaux d’obtenir les stratégies, les versions des stratégies, les rôles, les stratégies de rôle, les stratégies de rôles associées et la liste des rôles.
- `kms` : permet aux principaux de répertorier les alias.
- `logs` : permet aux principaux de décrire les flux de journaux, d’obtenir les événements du journal, de filtrer les événements du journal et de démarrer et arrêter dsessions Live Tail.
- `states`— Permet aux directeurs de décrire et de répertorier les machines AWS Step Functions d'état.
- `tag` : permet aux principaux d’obtenir des ressources en fonction de leurs balises.
- `xray`— Permet aux principaux d'obtenir des résumés de AWS X-Ray traces et de récupérer une liste de traces spécifiée par ID.

Pour plus d'informations sur cette politique, y compris le document de politique JSON et les versions de politique, consultez AWSLambda_ReadOnlyAccesslе Guide de référence des politiques AWS gérées.

AWS politique gérée : AWSLambda BasicExecutionRole

Cette politique accorde des autorisations pour télécharger des CloudWatch journaux dans Logs.

Vous pouvez attacher la stratégie `AWSLambdaBasicExecutionRole` à vos utilisateurs, groupes et rôles.

Pour plus d'informations sur cette politique, y compris le document de politique JSON et les versions de politique, consultez AWSLambdaBasicExecutionRolele Guide de référence des politiques AWS gérées.

AWS politique gérée : AWSLambda BasicDurableExecutionRolePolicy

Cette politique fournit des autorisations d'écriture aux CloudWatch journaux et read/write des autorisations d'exécution durable APIs utilisées par les fonctions durables Lambda. Cette politique fournit les autorisations essentielles requises pour les fonctions durables Lambda, qui utilisent l'exécution durable pour maintenir la progression et maintenir l'état d'un appel de fonction APIs à l'autre.

Vous pouvez attacher la stratégie `AWSLambdaBasicDurableExecutionRolePolicy` à vos utilisateurs, groupes et rôles.

Détails de l’autorisation

Cette politique inclut les autorisations suivantes :

- `logs`— Permet aux principaux de créer des groupes de journaux et des flux de journaux, et d'écrire les événements des journaux dans les CloudWatch journaux.
- `lambda`— Permet aux principaux de vérifier l'état d'exécution durable et de récupérer l'état d'exécution durable pour les fonctions durables Lambda.

Pour plus de détails sur cette politique, y compris la dernière version du document sur la politique JSON, consultez AWSLambdaBasicDurableExecutionRolePolicy dans le Guide de référence de la politique gérée par AWS .

AWS stratégie gérée : AWSLambda Dynamo Role DBExecution

Cette politique accorde des autorisations pour lire les enregistrements d'un flux Amazon DynamoDB et pour écrire dans Logs. CloudWatch

Vous pouvez attacher la stratégie `AWSLambdaDynamoDBExecutionRole` à vos utilisateurs, groupes et rôles.

Pour plus d'informations sur cette politique, y compris le document de politique JSON et les versions de politique, consultez AWSLambdaDynamo DBExecution Role dans le Guide de référence des politiques AWS gérées.

AWS politique gérée : AWSLambda ENIManagement Accès

Cette stratégie accorde les autorisations nécessaires pour créer, décrire et supprimer des interfaces réseau Elastic utilisées par une fonction Lambda compatible VPC.

Vous pouvez attacher la stratégie `AWSLambdaENIManagementAccess` à vos utilisateurs, groupes et rôles.

Pour plus d'informations sur cette politique, y compris le document de politique JSON et les versions de politique, consultez AWSLambdaENIManagementAccess dans le guide de référence des politiques AWS gérées.

AWS politique gérée : AWSLambdaInvocation-DynamoDB

Cette stratégie accorde un accès en lecture à Amazon DynamoDB Streams.

Vous pouvez attacher la stratégie `AWSLambdaInvocation-DynamoDB` à vos utilisateurs, groupes et rôles.

Pour plus d'informations sur cette politique, y compris le document de politique JSON et les versions de politique, consultez AWSLambdaInvocation-DynamoDBle Guide de référence des politiques AWS gérées.

AWS politique gérée : AWSLambda KinesisExecutionRole

Cette politique accorde l'autorisation de lire les événements d'un flux de données Amazon Kinesis et d'écrire dans Logs. CloudWatch

Vous pouvez attacher la stratégie `AWSLambdaKinesisExecutionRole` à vos utilisateurs, groupes et rôles.

Pour plus d'informations sur cette politique, y compris le document de politique JSON et les versions de politique, consultez AWSLambdaKinesisExecutionRolele Guide de référence des politiques AWS gérées.

AWS politique gérée : AWSLambda MSKExecution rôle

Cette politique autorise la lecture et l'accès aux enregistrements d'un cluster Amazon Managed Streaming for Apache Kafka, la gestion des interfaces réseau élastiques et l'écriture dans les CloudWatch journaux.

Vous pouvez attacher la stratégie `AWSLambdaMSKExecutionRole` à vos utilisateurs, groupes et rôles.

Pour plus d'informations sur cette politique, y compris le document de politique JSON et les versions de politique, voir AWSLambdaMSKExecutionRôle dans le guide de référence des politiques AWS gérées.

AWS politique gérée : AWSLambda rôle

Cette stratégie accorde des autorisations pour invoquer les fonctions Lambda.

Vous pouvez attacher la stratégie `AWSLambdaRole` à vos utilisateurs, groupes et rôles.

Pour plus d'informations sur cette politique, y compris le document de politique JSON et les versions de politique, voir AWSLambdaRôle dans le guide de référence des politiques AWS gérées.

AWS politique gérée : AWSLambda SQSQueue ExecutionRole

Cette politique accorde des autorisations pour lire et supprimer des messages d'une file d'attente Amazon Simple Queue Service, et accorde des autorisations d'écriture aux CloudWatch journaux.

Vous pouvez attacher la stratégie `AWSLambdaSQSQueueExecutionRole` à vos utilisateurs, groupes et rôles.

Pour plus d'informations sur cette politique, y compris le document de politique JSON et les versions de politique, consultez AWSLambdaSQSQueueExecutionRolele Guide de référence des politiques AWS gérées.

AWS politique gérée : AWSLambda VPCAccess ExecutionRole

Cette politique accorde des autorisations pour gérer des interfaces réseau élastiques au sein d'un Amazon Virtual Private Cloud et pour écrire dans CloudWatch Logs.

Vous pouvez attacher la stratégie `AWSLambdaVPCAccessExecutionRole` à vos utilisateurs, groupes et rôles.

Pour plus d'informations sur cette politique, y compris le document de politique JSON et les versions de politique, consultez AWSLambdaVPCAccessExecutionRolele Guide de référence des politiques AWS gérées.

AWS politique gérée : AWSLambda Géré EC2 ResourceOperator

Cette politique permet la gestion automatisée des instances Amazon Elastic Compute Cloud pour les fournisseurs de capacité Lambda. Il accorde des autorisations au service Lambda Scaler pour effectuer des opérations sur le cycle de vie des instances en votre nom.

Vous pouvez attacher la stratégie `AWSLambdaManagedEC2ResourceOperator` à vos utilisateurs, groupes et rôles.

Détails de l’autorisation

Cette politique inclut les autorisations suivantes :

- `ec2:RunInstances`— Permet à Lambda de lancer de nouvelles EC2 instances Amazon à condition que ec2 : soit ManagedResourceOperator égal à scaler.lambda.amazonaws.com et limite l'utilisation des AMI aux images appartenant à Amazon uniquement.
- `ec2:DescribeInstanceset ec2:DescribeInstanceStatus` — Permet à Lambda de surveiller l'état de l'instance et de récupérer les informations relatives à l'instance.
- `ec2:CreateTags`— Permet à Lambda de baliser les EC2 ressources Amazon à des fins de gestion et d'identification.
- `ec2:DescribeAvailabilityZones`— Permet à Lambda de visualiser les zones disponibles pour les décisions de placement des instances.
- `ec2:DescribeCapacityReservations`— Permet à Lambda de vérifier les réservations de capacité pour un placement optimal des instances.
- `ec2:DescribeInstanceTypeset ec2:DescribeInstanceTypeOfferings` — Permet à Lambda de passer en revue les types d'instances disponibles et leurs offres.
- `ec2:DescribeSubnets`— Permet à Lambda d'examiner les configurations de sous-réseau pour la planification du réseau.
- `ec2:DescribeSecurityGroups`— Permet à Lambda de récupérer les informations du groupe de sécurité pour la configuration de l'interface réseau.
- `ec2:CreateNetworkInterface`— Permet à Lambda de créer des interfaces réseau et de gérer les associations de sous-réseaux et de groupes de sécurité.
- `ec2:AttachNetworkInterface`— Permet à Lambda d'associer des interfaces réseau aux EC2 instances Amazon avec la condition `ec2:ManagedResourceOperator` égale à scaler.lambda.amazonaws.com.

Pour plus d'informations sur cette politique, y compris le document de politique JSON et les versions de politique, voir AWSLambdaManaged EC2 ResourceOperator dans le AWS Managed Policy Reference Guide.

AWS politique gérée : AWSLambda ServiceRolePolicy

Cette politique est attachée au rôle lié au service nommé pour permettre à Lambda AWSService RoleForLambda de mettre fin aux instances gérées dans le cadre des fournisseurs de capacité Lambda.

Détails de l’autorisation

Cette politique inclut les autorisations suivantes :

- `ec2:TerminateInstances`— Permet à Lambda de mettre fin à des EC2 instances à la condition que ec2 : ManagedResourceOperator soit égal à scaler.lambda.amazonaws.com.
- `ec2:DescribeInstanceStatus`et `ec2:DescribeInstances` — Permet à Lambda de décrire EC2 les instances.

Pour plus d'informations sur cette politique, consultez la section Utilisation de rôles liés à un service pour Lambda.

Mises à jour Lambda des politiques gérées AWS

Modifier	Description	Date
AWSLambdaGéré EC2 ResourceOperator — Nouvelle politique	Lambda a ajouté une nouvelle politique gérée pour permettre la gestion automatisée des EC2 instances Amazon pour les fournisseurs de capacité Lambda, permettant ainsi au service Scaler d'effectuer des opérations sur le cycle de vie des instances.	30 novembre 2025
AWSLambdaServiceRolePolicy : nouvelle politique	Lambda a ajouté une nouvelle politique gérée pour le rôle lié au service afin de permettre à Lambda de mettre fin aux instances gérées dans le cadre des fournisseurs de capacité Lambda.	30 novembre 2025
AWSLambda_FullAccess— Modification	Lambda a mis à jour la `AWSLambda_FullAccess` politique pour autoriser les actions `kms:DescribeKey`	30 novembre 2025

Modifier	Description	Date
	`etiam:CreateServiceLinkedRole` .	
AWSLambdaBasicDurableExecutionRolePolicy— Nouvelle politique gérée	Lambda a publié une nouvelle politique gérée `AWSLambdaBasicDurableExecutionRolePolicy` qui fournit des autorisations d'écriture aux CloudWatch journaux et des autorisations d'exécution durable APIs utilisées par read/write les fonctions durables de Lambda.	1er décembre 2025
AWSLambda_ReadOnlyAccesset AWSLambda_FullAccess— Modifier	Lambda a mis à jour les politiques `AWSLambda_ReadOnlyAccess` et `AWSLambda_FullAccess` pour autoriser les actions `logs:StartLiveTail` et `logs:StopLiveTail` .	17 mars 2025
AWSLambdaVPCAccessExecutionRole— Modification	Lambda a mis à jour la politique `AWSLambdaVPCAccessExecutionRole` pour autoriser l'action `ec2:DescribeSubnets` .	5 janvier 2024
AWSLambda_ReadOnlyAccess— Modification	Lambda a mis à jour la `AWSLambda_ReadOnlyAccess` politique pour autoriser les principaux à répertorier les piles. CloudFormation	27 juillet 2023

Modifier	Description	Date
AWS Lambda a commencé à suivre les modifications	AWS Lambda a commencé à suivre les modifications apportées AWS à ses politique s gérées.	27 juillet 2023

Résolution des problèmes d'identité et d'accès avec AWS Lambda

Utilisez les informations suivantes pour identifier et résoudre les problèmes courants que vous pouvez rencontrer lorsque vous travaillez avec Lambda et IAM.

Rubriques

- Je ne suis pas autorisé à effectuer une action dans Lambda
- Je ne suis pas autorisé à exécuter : iam:PassRole
- Je veux autoriser des personnes extérieures à mon Compte AWS à accéder à mes ressources Lambda

Je ne suis pas autorisé à effectuer une action dans Lambda

Si vous recevez une erreur qui indique que vous n'êtes pas autorisé à effectuer une action, vos politiques doivent être mises à jour afin de vous permettre d'effectuer l'action.

L'exemple d'erreur suivant se produit quand l'utilisateur IAM `mateojackson` tente d'utiliser la console pour afficher des informations détaillées sur une ressource *`my-example-widget`* fictive, mais ne dispose pas des autorisations `lambda:`*`GetWidget`* fictives.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 lambda:GetWidget on resource: my-example-widget
```

Dans ce cas, la politique qui s'applique à l'utilisateur `mateojackson` doit être mise à jour pour autoriser l'accès à la ressource *`my-example-widget`* à l'aide de l'action `lambda:`*`GetWidget`*.

Si vous avez encore besoin d'aide, contactez votre administrateur AWS. Votre administrateur vous a fourni vos informations d'identification de connexion.

Je ne suis pas autorisé à exécuter : iam:PassRole

Si vous recevez une erreur selon laquelle vous n'êtes pas autorisé à exécuter l'action `iam:PassRole`, vos politiques doivent être mises à jour pour vous permettre de transmettre un rôle à Lambda.

Certains Services AWS vous permettent de transmettre un rôle existant à ce service, au lieu de créer un nouveau rôle de service ou rôle lié à un service. Pour ce faire, un utilisateur doit disposer des autorisations nécessaires pour transmettre le rôle au service.

L'exemple d'erreur suivant se produit quand un utilisateur IAM nommé `marymajor` tente d'utiliser la console pour exécuter une action dans Lambda. Toutefois, l'action nécessite que le service ait des autorisations accordées par un rôle de service. Mary ne dispose pas des autorisations nécessaires pour transférer le rôle au service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

Dans ce cas, les politiques de Mary doivent être mises à jour pour lui permettre d'exécuter l'action `iam:PassRole`.

Si vous avez encore besoin d'aide, contactez votre administrateur AWS. Votre administrateur vous a fourni vos informations d'identification de connexion.

Je veux autoriser des personnes extérieures à mon Compte AWS à accéder à mes ressources Lambda

Vous pouvez créer un rôle que les utilisateurs provenant d'autres comptes ou les personnes extérieures à votre organisation pourront utiliser pour accéder à vos ressources. Vous pouvez spécifier qui est autorisé à assumer le rôle. Pour les services qui prennent en charge les politiques basées sur les ressources ou les listes de contrôle d'accès (ACL), vous pouvez utiliser ces politiques pour donner l'accès à vos ressources.

Pour en savoir plus, consultez les éléments suivants :

- Pour savoir si Lambda prend en charge ces fonctions, consultez Fonctionnement d'AWS Lambda avec IAM.
- Pour savoir comment octroyer l'accès à vos ressources à des Comptes AWS dont vous êtes propriétaire, consultez la section Fournir l'accès à un utilisateur IAM dans un autre Compte AWS que vous possédez dans le Guide de l'utilisateur IAM.

- Pour savoir comment octroyer l'accès à vos ressources à des tiers Comptes AWS, consultez Fournir l'accès aux Comptes AWS appartenant à des tiers dans le Guide de l'utilisateur IAM.
- Pour savoir comment fournir un accès par le biais de la fédération d'identité, consultez Fournir un accès à des utilisateurs authentifiés en externe (fédération d'identité) dans le Guide de l'utilisateur IAM.
- Pour en savoir plus sur la différence entre l'utilisation des rôles et des politiques basées sur les ressources pour l'accès intercompte, consultez Accès intercompte aux ressources dans IAM dans le Guide de l'utilisateur IAM.

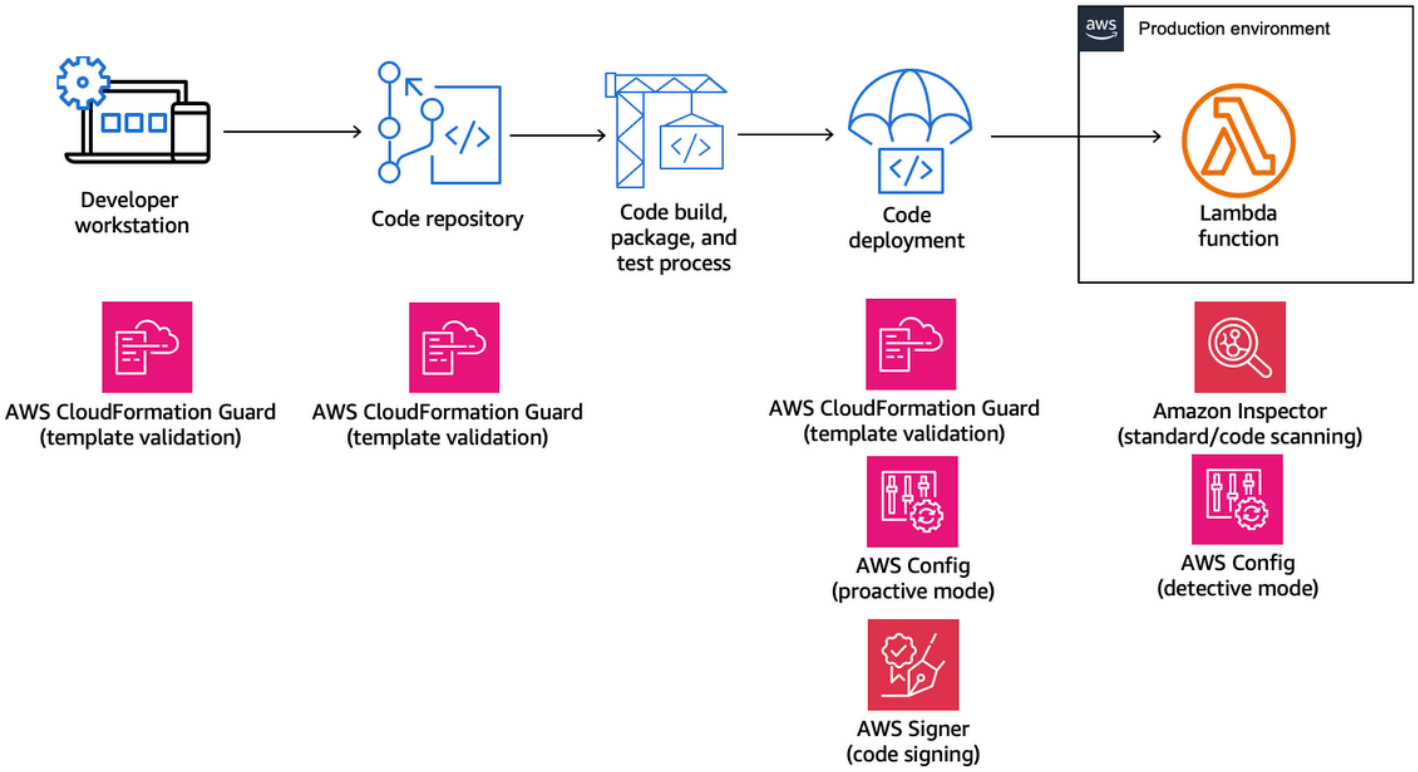
Création d'une stratégie de gouvernance pour les couches et les fonctions Lambda

Pour créer et déployer des applications cloud natives sans serveur, vous devez garantir agilité et rapidité de mise sur le marché grâce à une gouvernance et à des barrières de protection appropriés. Vous définissez des priorités au niveau de l'entreprise, en mettant peut-être l'accent sur l'agilité comme priorité absolue, ou en insistant sur l'aversion au risque par le biais de la gouvernance, de barrières de protection et de contrôles. En réalité, vous n'aurez pas une stratégie « l'un ou l'autre », mais une stratégie « et » qui équilibre à la fois agilité et barrières de protection dans le cycle de vie de votre développement logiciel. Quelle que soit la place de ces exigences dans le cycle de vie de votre entreprise, les capacités de gouvernance sont susceptibles de devenir une exigence de mise en œuvre dans vos processus et chaînes d'outils.

Voici quelques exemples de contrôles de gouvernance qu'une organisation peut mettre en œuvre pour Lambda :

- Les fonctions Lambda ne doivent pas être publiquement accessibles.
- Les fonctions Lambda doivent être associées à un VPC.
- Les fonctions Lambda ne doivent pas utiliser d'environnements d'exécution obsolètes.
- Les fonctions Lambda doivent être étiquetées avec un ensemble de balises obligatoires.
- Les couches Lambda ne doivent pas être accessibles en dehors de l'organisation.
- Les fonctions Lambda associées à un groupe de sécurité doivent avoir des balises correspondantes entre la fonction et le groupe de sécurité.
- Les fonctions Lambda associées à une couche doivent utiliser une version approuvée
- Les variables d'environnement Lambda doivent être chiffrées au repos avec une clé gérée par le client.

Le schéma suivant est un exemple de stratégie de gouvernance approfondie qui met en œuvre des contrôles et des politiques tout au long du processus de développement et de déploiement du logiciel :



Les rubriques suivantes expliquent comment implémenter des contrôles pour développer et déployer des fonctions Lambda dans votre organisation, à la fois pour la start-up et pour l'entreprise. Votre organisation possède peut-être déjà des outils en place. Les rubriques suivantes adoptent une approche modulaire de ces contrôles, afin que vous puissiez sélectionner les composants dont vous avez réellement besoin.

Rubriques

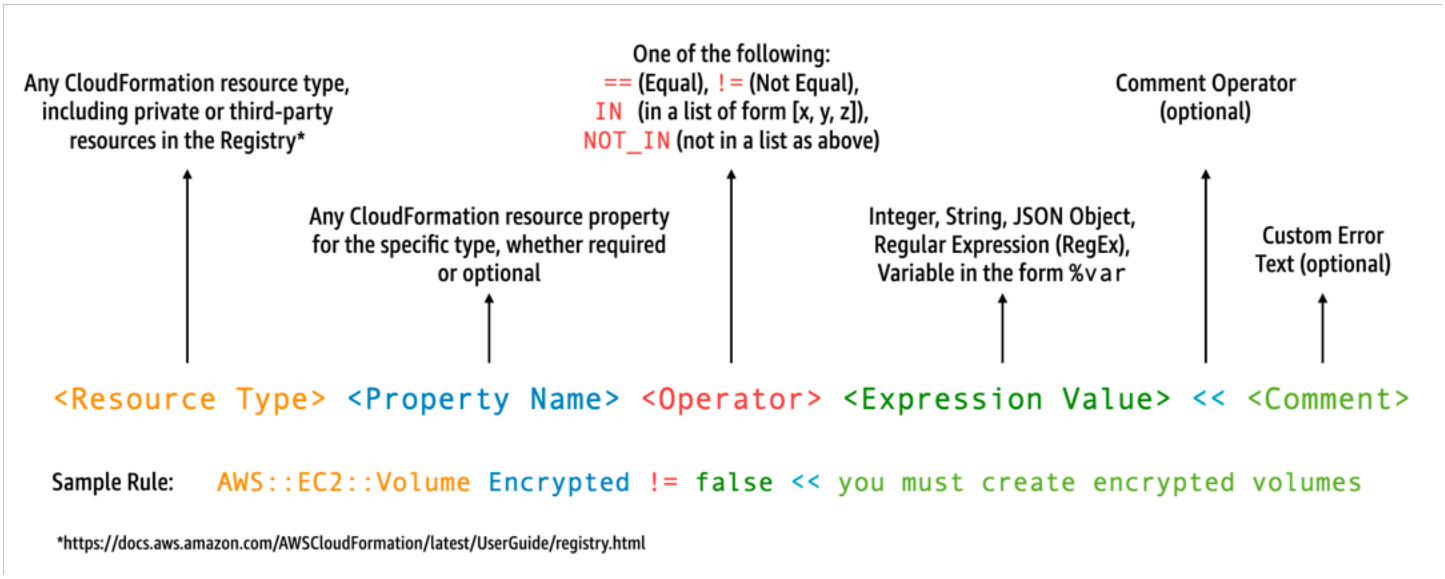
- Contrôles proactifs pour Lambda avec AWS CloudFormation Guard
- Mettez en œuvre des contrôles préventifs pour Lambda avec AWS Config
- Détection des déploiements et des configurations Lambda non conformes avec AWS Config
- Signature de code Lambda avec AWS Signer
- Automatisation des évaluations de sécurité pour Lambda avec Amazon Inspector
- Mettre en œuvre l'observabilité pour la sécurité et la conformité Lambda

Contrôles proactifs pour Lambda avec AWS CloudFormation Guard

AWS CloudFormation Guard est un outil d'évaluation des politiques en tant que code open source à usage général. Cela peut être utilisé à des fins de gouvernance préventive et de conformité en validant les modèles d'infrastructure en tant que code (IaC) et les compositions de services par rapport aux règles de politique. Ces règles peuvent être personnalisées en fonction des exigences de votre équipe ou de votre organisation. Pour les fonctions Lambda, les règles Guard peuvent être utilisées pour contrôler la création de ressources et les mises à jour de configuration en définissant les paramètres de propriété requis lors de la création ou de la mise à jour d'une fonction Lambda.

Les administrateurs de conformité définissent la liste des contrôles et des politiques de gouvernance nécessaires au déploiement et à la mise à jour des fonctions Lambda. Les administrateurs de plateforme mettent en œuvre les contrôles dans les pipelines CI/CD, sous forme de webhooks de validation préalable à la validation avec des référentiels de code, et fournissent aux développeurs des outils en ligne de commande pour valider les modèles et le code sur les postes de travail locaux. Les développeurs créent du code, valident les modèles à l'aide d'outils de ligne de commande, puis valident le code dans des référentiels, qui sont ensuite automatiquement validés via les pipelines CI/CD avant le déploiement dans un environnement AWS.

Guard vous permet d'écrire vos règles et d'implémenter vos contrôles dans un langage spécifique au domaine comme suit.



Supposons, par exemple, que vous souhaitiez vous assurer que les développeurs choisissent uniquement les derniers environnements d'exécution. Vous pouvez spécifier deux politiques différentes, l'une pour identifier les environnements d'exécution déjà obsolètes et l'autre pour

identifier les environnements d'exécution qui le seront bientôt. Pour cela, vous pouvez écrire le fichier `etc/rules.guard` suivant :

```
let lambda_functions = Resources.*[
    Type == "AWS::Lambda::Function"
]

rule lambda_already_deprecated_runtime when %lambda_functions !empty {
    %lambda_functions {
        Properties {
            when Runtime exists {
                Runtime !in ["dotnetcore3.1", "nodejs12.x", "python3.6", "python2.7",
 "dotnet5.0", "dotnetcore2.1", "ruby2.5", "nodejs10.x", "nodejs8.10", "nodejs4.3",
 "nodejs6.10", "dotnetcore1.0", "dotnetcore2.0", "nodejs4.3-edge", "nodejs"] <<Lambda
 function is using a deprecated runtime.>>
            }
        }
    }
}

rule lambda_soon_to_be_deprecated_runtime when %lambda_functions !empty {
    %lambda_functions {
        Properties {
            when Runtime exists {
                Runtime !in ["nodejs16.x", "nodejs14.x", "python3.7", "java8",
 "dotnet7", "go1.x", "ruby2.7", "provided"] <<Lambda function is using a runtime that
 is targeted for deprecation.>>
            }
        }
    }
}
```

Supposons maintenant que vous écriviez le modèle `iac/lambda.yaml` CloudFormation suivant qui définit une fonction Lambda :

```
  Fn:
    Type: AWS::Lambda::Function
    Properties:
      Runtime: python3.7
      CodeUri: src
      Handler: fn.handler
      Role: !GetAtt FnRole.Arn
      Layers:
```

```
        - arn:aws:lambda:us-east-1:111122223333:layer:LambdaInsightsExtension:35
```

Après avoir installé l'utilitaire Guard, validez votre modèle :

```
cfn-guard validate --rules etc/rules.guard --data iac/lambda.yaml
```

Le résultat se présente comme suit :

```
lambda.yaml Status = FAIL
FAILED rules
rules.guard/lambda_soon_to_be_deprecated_runtime
---
Evaluating data lambda.yaml against rules rules.guard
Number of non-compliant resources 1
Resource = Fn {
  Type      = AWS::Lambda::Function
  Rule = lambda_soon_to_be_deprecated_runtime {
    ALL {
      Check =  Runtime not IN
 ["nodejs16.x","nodejs14.x","python3.7","java8","dotnet7","go1.x","ruby2.7","provided"]
 {
       ComparisonError {
         Message          = Lambda function is using a runtime that is targeted for
 deprecation.
         Error            = Check was not compliant as property [/Resources/
Fn/Properties/Runtime[L:88,C:15]] was not present in [(resolved, Path=[L:0,C:0]
 Value=["nodejs16.x","nodejs14.x","python3.7","java8","dotnet7","go1.x","ruby2.7","provided"])]
       }
         PropertyPath    = /Resources/Fn/Properties/Runtime[L:88,C:15]
         Operator        = NOT IN
         Value           = "python3.7"
         ComparedWith    =
 [["nodejs16.x","nodejs14.x","python3.7","java8","dotnet7","go1.x","ruby2.7","provided"]]
         Code:
              86.  Fn:
              87.    Type: AWS::Lambda::Function
              88.    Properties:
              89.      Runtime: python3.7
              90.      CodeUri: src
              91.      Handler: fn.handler

      }
    }
```

```
  }
}
```

Guard permet à vos développeurs de savoir depuis leur poste de travail local qu'ils doivent mettre à jour le modèle afin d'utiliser un environnement d'exécution autorisé par l'organisation. Cela se produit avant la validation dans un référentiel de code et avant l'échec des vérifications dans un pipeline CI/CD. Vos développeurs reçoivent ainsi des informations sur la manière de développer des modèles conformes et de consacrer leur temps à l'écriture de code apportant une valeur commerciale.
Ce contrôle peut être appliqué sur le poste de travail du développeur local, dans un webhook de validation préalable à la validation et/ou dans le pipeline CI/CD avant le déploiement.

Mises en garde

Si vous utilisez des modèles AWS Serverless Application Model (AWS SAM) pour définir des fonctions Lambda, sachez que vous devez mettre à jour la règle Guard pour rechercher le type de ressource `AWS::Serverless::Function` comme suit.

```
let lambda_functions = Resources.*[
    Type == "AWS::Serverless::Function"
]
```

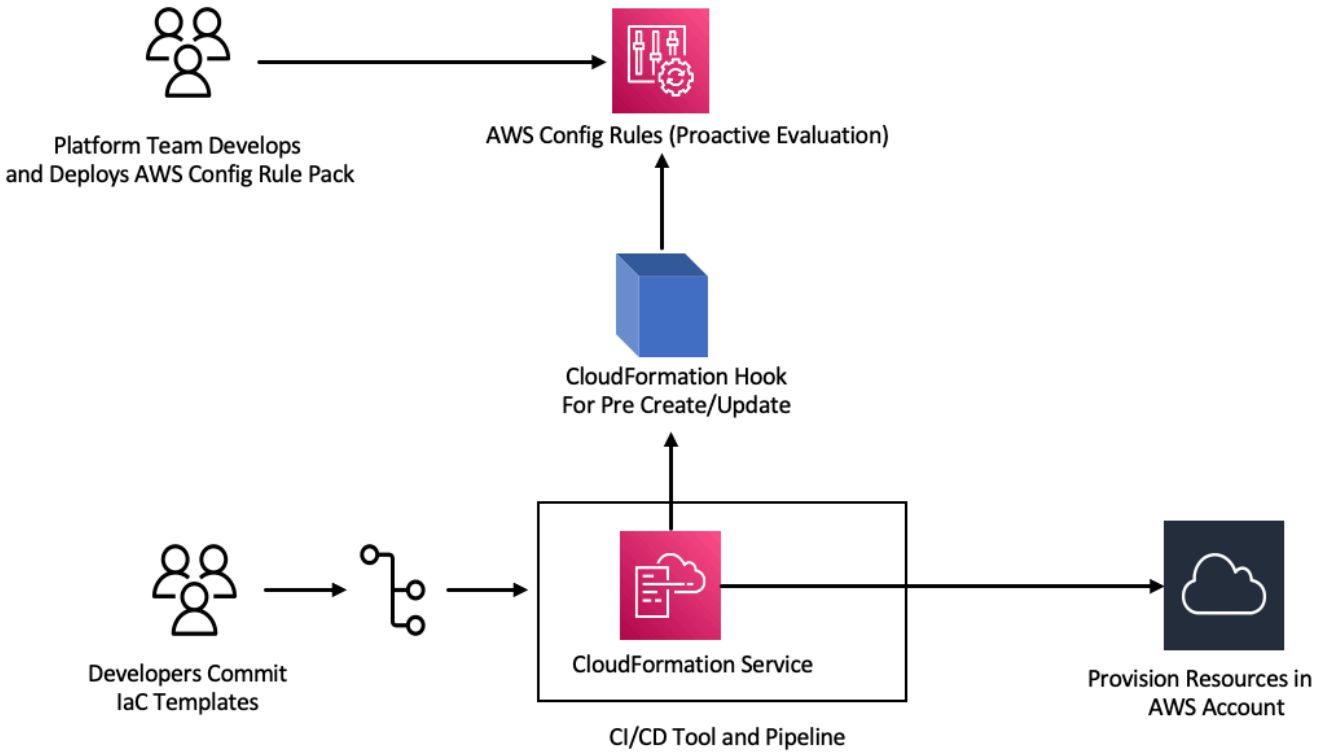
Guard s'attend également à ce que les propriétés soient incluses dans la définition de la ressource. Dans le même temps, les modèles AWS SAM permettent de spécifier les propriétés dans une section Globals distincte. Les propriétés définies dans la section Globals ne sont pas validées par vos règles Guard.

Comme indiqué dans la documentation de dépannage de Guard, sachez que Guard ne prend pas en charge les formulaires intrinsèques abrégés tels que `!GetAtt` ou `!Sub` et nécessite plutôt l'utilisation des formulaires étendus : `Fn::GetAtt` et. `Fn::Sub` (L'exemple précédent n'évaluant pas la propriété Role, la forme abrégée intrinsèque a été utilisée pour des raisons de simplicité.)

Mettez en œuvre des contrôles préventifs pour Lambda avec AWS Config

Il est essentiel de garantir la conformité de vos applications sans serveur le plus tôt possible dans le processus de développement. Dans cette rubrique, nous expliquons comment mettre en œuvre des contrôles préventifs à l'aide de AWS Config. Cela vous permet de mettre en œuvre des contrôles de conformité plus tôt dans le processus de développement et de mettre en œuvre les mêmes contrôles dans vos CI/CD pipelines. Cela permet également de standardiser vos contrôles dans un référentiel de règles géré de manière centralisée afin que vous puissiez appliquer vos contrôles de manière cohérente sur l'ensemble de vos AWS comptes.

Supposons, par exemple, que vos administrateurs de conformité aient défini une exigence garantissant que toutes les fonctions Lambda incluent AWS X-Ray le traçage. Grâce AWS Config au mode proactif, vous pouvez effectuer des contrôles de conformité sur les ressources de vos fonctions Lambda avant le déploiement, ce qui réduit le risque de déploiement de fonctions Lambda mal configurées et fait gagner du temps aux développeurs en leur fournissant plus rapidement des informations sur l'infrastructure sous forme de modèles de code. Voici une visualisation du flux pour les contrôles préventifs avec AWS Config :



Envisagez d'exiger que le suivi soit activé pour toutes les fonctions Lambda. En réponse, l'équipe de la plateforme identifie la nécessité d'une AWS Config règle spécifique pour s'exécuter de manière proactive sur tous les comptes. Cette règle signale toute fonction Lambda dont la configuration de suivi X-Ray n'est pas configurée comme une ressource non conforme. L'équipe développe une règle, l'intègre dans un pack de conformité et déploie le pack de conformité sur tous les comptes afin de garantir que tous les AWS comptes de l'organisation appliquent ces contrôles de manière uniforme. Vous pouvez écrire la règle dans la syntaxe AWS CloudFormation Guard 2.x.x, qui prend la forme suivante :

```
rule name when condition { assertion }
```

Voici un exemple de règle Guard qui vérifie que le suivi est activé pour les fonctions Lambda :

```
rule lambda_tracing_check {
  when configuration.tracingConfig exists {
      configuration.tracingConfig.mode == "Active"
  }
}
```

L'équipe de la plateforme prend des mesures supplémentaires en imposant que chaque AWS CloudFormation déploiement invoque un hook de pré-création/mise à jour. Elle assume l'entière responsabilité du développement de ce hook et de la configuration du pipeline, du renforcement du contrôle centralisé des règles de conformité et du maintien de leur application cohérente dans tous les déploiements. Pour développer, empaqueter et enregistrer un hook, consultez la documentation AWS CloudFormation Developing Hooks in the CloudFormation Command Line Interface (CFN-CLI). Vous pouvez utiliser la CloudFormation CLI pour créer le projet hook :

```
cfn init
```

Cette commande vous demande des informations de base sur votre projet hook et crée un projet contenant les fichiers suivants :

```
README.md
<hook-name>.json
rpdk.log
src/handler.py
template.yml
hook-role.yaml
```

En tant que développeur de hooks, vous devez ajouter le type de ressource cible souhaité dans le fichier `<hook-name>.json` de configuration. Dans la configuration ci-dessous, un hook est configuré pour s'exécuter avant qu'une fonction Lambda ne soit créée à l'aide de. CloudFormation Vous pouvez également ajouter des gestionnaires `preUpdate` et des actions `preDelete` similaires.

```
    "handlers": {
        "preCreate": {
            "targetNames": [
                "AWS::Lambda::Function"
            ],
            "permissions": []
        }
    }
```

Vous devez également vous assurer que le CloudFormation hook dispose des autorisations appropriées pour appeler le AWS Config APIs. Vous pouvez le faire en mettant à jour le fichier de définition de rôle nommé `hook-role.yaml`. Le fichier de définition du rôle possède par défaut la politique de confiance suivante, qui permet CloudFormation d'assumer le rôle.

```
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service:
                - hooks.cloudformation.amazonaws.com
                - resources.cloudformation.amazonaws.com
```

Pour permettre à ce hook d'appeler config APIs, vous devez ajouter les autorisations suivantes à la déclaration Policy. Ensuite, vous soumettez le projet hook à l'aide de la `cfn submit` commande, où vous CloudFormation créez un rôle avec les autorisations requises.

```
      Policies:
        - PolicyName: HookTypePolicy
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              - Effect: Allow
                Action:
                  - "config:Describe*"
                  - "config:Get*"
```

```
                - "config:List*"
                - "config:SelectResourceConfig"
              Resource: "*
```

Ensuite, vous devez écrire une fonction Lambda dans un `src/handler.py` fichier. Dans ce fichier, vous trouverez les méthodes nommées `preCreate`, `preUpdate` et `preDelete` déjà créées lorsque vous avez lancé le projet. Votre objectif est d'écrire une fonction commune et réutilisable qui appelle l' AWS Config `StartResourceEvaluation`API en mode proactif à l'aide du AWS SDK pour Python (Boto3). Cet appel d'API prend les propriétés des ressources en entrée et évalue la ressource par rapport à la définition de la règle.

```
def validate_lambda_tracing_config(resource_type, function_properties:
 MutableMapping[str, Any]) -> ProgressEvent:
  LOG.info("Fetching proactive data")
  config_client = boto3.client('config')
  resource_specs = {
      'ResourceId': 'MyFunction',
      'ResourceType': resource_type,
      'ResourceConfiguration': json.dumps(function_properties),
      'ResourceConfigurationSchemaType': 'CFN_RESOURCE_SCHEMA'
  }
  LOG.info("Resource Specifications:", resource_specs)
  eval_response = config_client.start_resource_evaluation(EvaluationMode='PROACTIVE',
 ResourceDetails=resource_specs, EvaluationTimeout=60)
  ResourceEvaluationId = eval_response.ResourceEvaluationId
  compliance_response =
 config_client.get_compliance_details_by_resource(ResourceEvaluationId=ResourceEvaluationId)
  LOG.info("Compliance Verification:",
 compliance_response.EvaluationResults[0].ComplianceType)
  if "NON_COMPLIANT" == compliance_response.EvaluationResults[0].ComplianceType:
      return ProgressEvent(status=OperationStatus.FAILED, message="Lambda function
 found with no tracing enabled : FAILED", errorCode=HandlerErrorCode.NonCompliant)
  else:
      return ProgressEvent(status=OperationStatus.SUCCESS, message="Lambda function
 found with tracing enabled : PASS.")
```

Vous pouvez maintenant appeler la fonction commune depuis le gestionnaire pour le hook de pré-création. Voici un exemple de gestionnaire :

```
@hook.handler(HookInvocationPoint.CREATE_PRE_PROVISION)
def pre_create_handler(
        session: Optional[SessionProxy],
```

```
        request: HookHandlerRequest,
        callback_context: MutableMapping[str, Any],
        type_configuration: TypeConfigurationModel
) -> ProgressEvent:
    LOG.info("Starting execution of the hook")
    target_name = request.hookContext.targetName
    LOG.info("Target Name:", target_name)
    if "AWS::Lambda::Function" == target_name:
        return validate_lambda_tracing_config(target_name,
            request.hookContext.targetModel.get("resourceProperties")
        )
    else:
        raise exceptions.InvalidRequest(f"Unknown target type: {target_name}")
```

Après cette étape, vous pouvez enregistrer le hook et le configurer pour écouter tous les événements de création de AWS Lambda fonctions.

Un développeur prépare le modèle d'infrastructure en tant que code (IaC) pour un microservice sans serveur à l'aide de Lambda. Cette préparation inclut le respect des normes internes, suivi de tests locaux et de validation du modèle dans le référentiel. Voici un exemple de modèle IaC :

```
  MyLambdaFunction:
  Type: 'AWS::Lambda::Function'
  Properties:
    Handler: index.handler
    Role: !GetAtt LambdaExecutionRole.Arn
    FunctionName: MyLambdaFunction
    Code:
      ZipFile: |
        import json

        def handler(event, context):
            return {
                'statusCode': 200,
                'body': json.dumps('Hello World!')
            }
    Runtime: python3.14
    TracingConfig:
        Mode: PassThrough
    MemorySize: 256
    Timeout: 10
```

Dans le cadre du CI/CD processus, lorsque le CloudFormation modèle est déployé, le CloudFormation service invoque le hook de pré-création/mise à jour juste avant le provisionnement du type de ressource. `AWS::Lambda::Function` Le hook utilise des AWS Config règles exécutées en mode proactif pour vérifier que la configuration de la fonction Lambda inclut la configuration de suivi obligatoire. La réponse du hook détermine l'étape suivante. S'il est conforme, le hook signale le succès et CloudFormation procède à la mise à disposition des ressources. Dans le cas contraire, le déploiement de la CloudFormation pile échoue, le pipeline s'arrête immédiatement et le système enregistre les détails pour un examen ultérieur. Les notifications de conformité sont envoyées vers les parties prenantes concernées.

Vous pouvez trouver les success/fail informations relatives au hook dans la CloudFormation console :

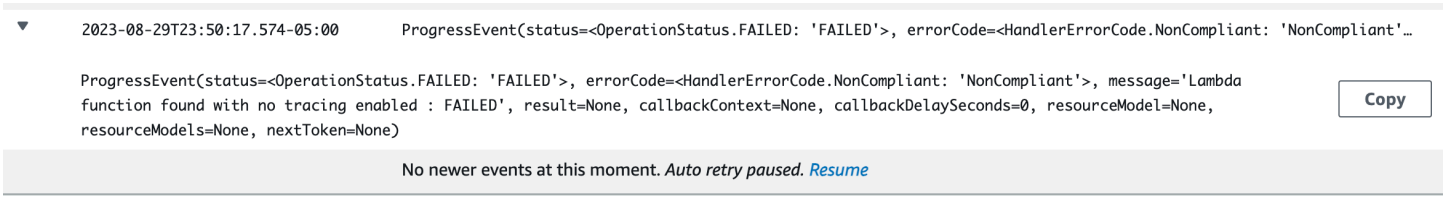
Stack info | Events | Resources | Outputs | Parameters | Template | Change sets

Events (19)

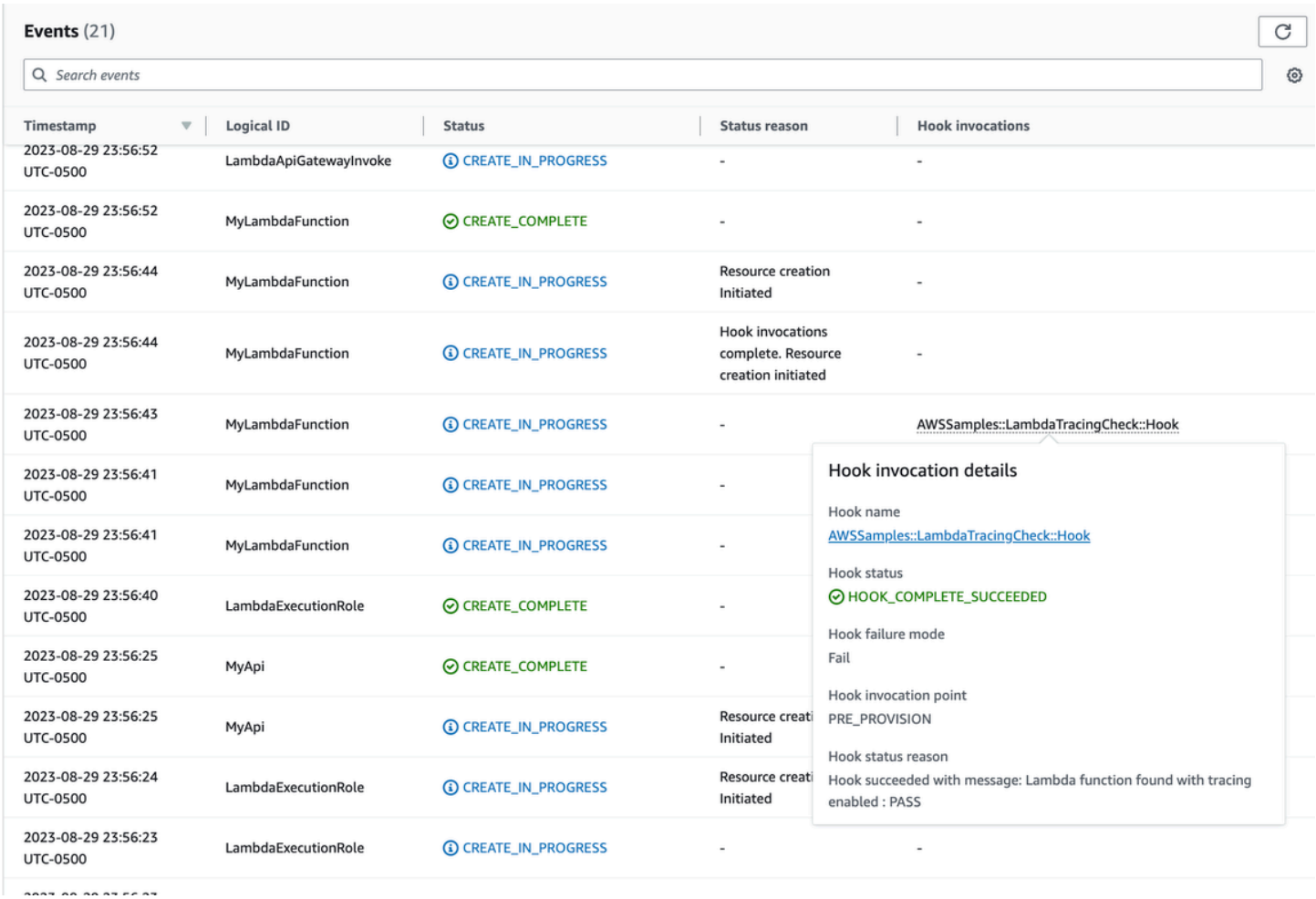
Search events

Timestamp	Logical ID	Status	Status reason	Hook invocations
2023-08-29 23:50:23 UTC-0500	HookTestStack	ROLLBACK_COMPLETE	-	-
2023-08-29 23:50:22 UTC-0500	LambdaExecutionRole	DELETE_COMPLETE	-	-
2023-08-29 23:50:21 UTC-0500	MyApi	DELETE_COMPLETE	-	-
2023-08-29 23:50:20 UTC-0500	LambdaExecutionRole	DELETE_IN_PROGRESS	-	-
2023-08-29 23:50:20 UTC-0500	MyLambdaFunction	DELETE_COMPLETE	-	-
2023-08-29 23:50:20 UTC-0500	MyApi	DELETE_IN_PROGRESS	-	-
2023-08-29 23:50:18 UTC-0500	HookTestStack	ROLLBACK_IN_PROGRESS	The following resource(s) failed to create: [MyLambdaFunction]. Rollback requested by user.	-
2023-08-29 23:50:17 UTC-0500	MyLambdaFunction	CREATE_FAILED	The following hook(s) failed: [AWSSamples::LambdaTracingCheck::Hook]	-
2023-08-29 23:50:17 UTC-0500	MyLambdaFunction	CREATE_IN_PROGRESS	-	AWSSamples::LambdaTracingCheck::Hook
2023-08-29 23:50:16 UTC-0500	MyLambdaFunction	CREATE_IN_PROGRESS	-	AWSSamples::LambdaTracingCheck::Hook
2023-08-29 23:50:15 UTC-0500	MyLambdaFunction	CREATE_IN_PROGRESS	-	-
2023-08-29 23:50:14 UTC-0500	LambdaExecutionRole	CREATE_COMPLETE	-	-
2023-08-29 23:49:59 UTC-0500	MyApi	CREATE_COMPLETE	-	-
2023-08-29 23:49:59 UTC-0500	MyApi	CREATE_IN_PROGRESS	Resource creation Initiated	-
2023-08-29 23:49:58 UTC-0500	LambdaExecutionRole	CREATE_IN_PROGRESS	Resource creation Initiated	-
2023-08-29 23:49:58 UTC-0500	LambdaExecutionRole	CREATE_IN_PROGRESS	-	-
2023-08-29 23:49:58 UTC-0500	MyApi	CREATE_IN_PROGRESS	-	-
2023-08-29 23:49:55 UTC-0500	HookTestStack	CREATE_IN_PROGRESS	User Initiated	-
2023-08-29 23:49:50 UTC-0500	HookTestStack	REVIEW_IN_PROGRESS	User Initiated	-

Si les journaux sont activés pour votre CloudFormation hook, vous pouvez capturer le résultat de l'évaluation du hook. Voici un exemple de journal pour un hook dont le statut a échoué, indiquant que X-Ray n'est pas activé sur la fonction Lambda :



Si le développeur choisit de modifier le modèle IaC pour mettre à jour la valeur `TracingConfig Mode`sur `Active` et le redéployer, le hook s'exécute correctement et la pile poursuit la création de la ressource Lambda.



Ainsi, vous pouvez mettre en œuvre des contrôles préventifs en mode proactif lors du développement et du déploiement de ressources sans serveur dans vos AWS comptes. AWS Config En intégrant des AWS Config règles dans le CI/CD pipeline, vous pouvez identifier et éventuellement bloquer

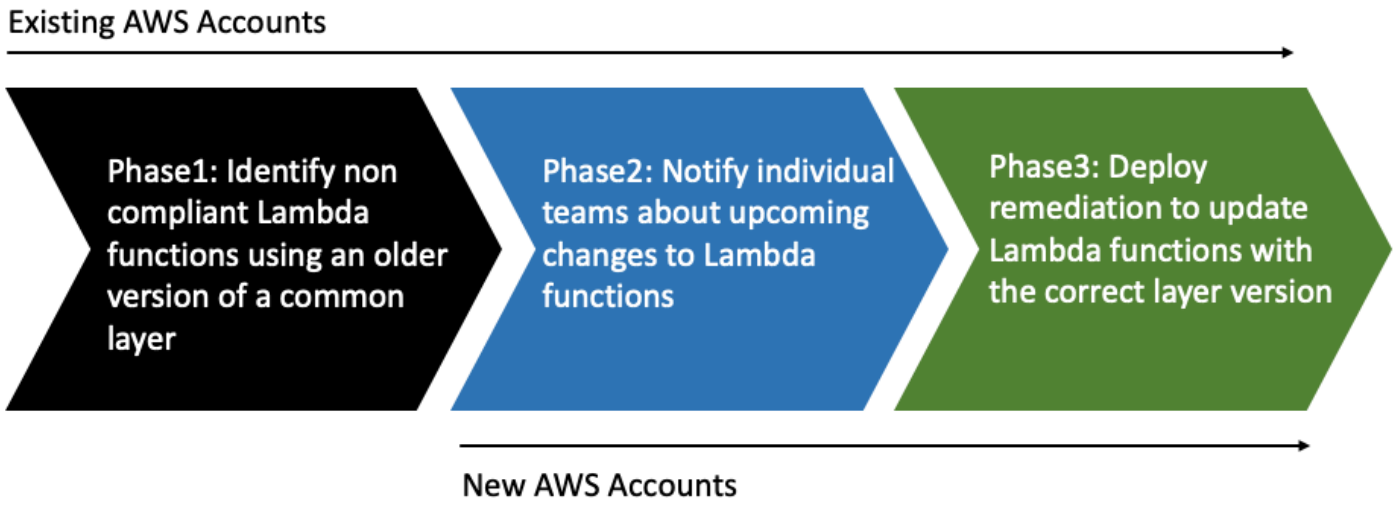
les déploiements de ressources non conformes, tels que les fonctions Lambda dépourvues de configuration de suivi active. Cela garantit que seules les ressources conformes aux dernières politiques de gouvernance sont déployées dans vos AWS environnements.

Détection des déploiements et des configurations Lambda non conformes avec AWS Config

Outre l'évaluation proactive, AWS Config peut également effectuer une détection réactive des déploiements de ressources et des configurations non conformes à vos politiques de gouvernance. Cela est important car les politiques de gouvernance évoluent au fur et à mesure que votre organisation apprend et met en œuvre de nouvelles meilleures pratiques.

Imaginons un scénario dans lequel vous définissez une toute nouvelle politique lors du déploiement ou de la mise à jour des fonctions Lambda : toutes les fonctions Lambda doivent toujours utiliser une version de couche Lambda spécifique et approuvée. Vous pouvez configurer AWS Config pour surveiller les fonctions nouvelles ou mises à jour pour les configurations de couches. Si AWS Config détecte une fonction qui n'utilise pas une version de couche approuvée, il signale la fonction comme ressource non conforme. Vous pouvez éventuellement configurer AWS Config pour corriger automatiquement la ressource en spécifiant une action de correction à l'aide d'un document d'automatisation AWS Systems Manager. Par exemple, vous pouvez écrire un document d'automatisation en Python à l'aide de AWS SDK pour Python (Boto3), qui met à jour la fonction non conforme pour qu'elle pointe vers la version de couche approuvée. Ainsi, AWS Config sert à la fois de détection et de contrôle correctif, automatisant la gestion de la conformité.

Décomposons ce processus en trois phases de mise en œuvre importantes :



Phase 1 : identification des ressources d'accès

Commencez par activer AWS Config sur tous vos comptes et par le configurer pour enregistrer les fonctions AWS Lambda. Cela permet à AWS Config d'observer quand les fonctions Lambda sont

créées ou mises à jour. Vous pouvez ensuite configurer des règles de stratégie personnalisées pour vérifier les violations de politique spécifiques, en utilisant la syntaxe AWS CloudFormation Guard. Les règles de garde prennent la forme générale suivante :

```
rule name when condition { assertion }
```

Vous trouverez ci-dessous un exemple de règle qui vérifie qu'une couche n'est pas définie sur une ancienne version de couche :

```
rule desiredlayer when configuration.layers !empty {
    some configuration.layers[*].arn != CONFIG_RULE_PARAMETERS.OldLayerArn
}
```

Découvrons la syntaxe et la structure des règles :

- Nom de la règle : le nom de la règle est `desiredlayer` dans l'exemple fourni.
- Condition : cette clause spécifie la condition dans laquelle la règle doit être vérifiée. Dans l'exemple fourni, la condition est `configuration.layers !empty`. Cela signifie que la ressource ne doit être évaluée que lorsque la propriété `layers` de la configuration n'est pas vide.
- Assertion : après la clause `when`, une assertion détermine ce que la règle vérifie. L'assertion `some configuration.layers[*].arn != CONFIG_RULE_PARAMETERS.OldLayerArn` vérifie si l'un des ARN de la couche Lambda ne correspond pas à la valeur `OldLayerArn`. S'ils ne correspondent pas, l'assertion est vraie et la règle est acceptée ; sinon, elle échoue.

`CONFIG_RULE_PARAMETERS` est un ensemble spécial de paramètres configuré avec la règle AWS Config. Dans ce cas, `OldLayerArn` est un paramètre dans `CONFIG_RULE_PARAMETERS`. Cela permet aux utilisateurs de fournir une valeur d'ARN spécifique qu'ils considèrent comme ancienne ou obsolète, puis la règle vérifie si des fonctions Lambda utilisent cet ancien ARN.

Phase 2 : Visualisation et conception

AWS Config collecte des données de configuration et les stocke dans des compartiments Amazon Simple Storage Service (Amazon S3). Vous pouvez utiliser Amazon Athena pour interroger ces données directement depuis vos compartiments S3. Avec Athena, vous pouvez agréger ces données au niveau de l'organisation, en générant une vue globale de la configuration de vos ressources sur l'ensemble de vos comptes. Pour configurer l'agrégation des données de configuration des ressources, consultez la section Visualisation AWS Config des données à l'aide d'Athena et d'Amazon Quick Suite sur le blog AWS Cloud Operations and Management.

Voici un exemple de requête Athena pour identifier toutes les fonctions Lambda à l'aide d'un ARN de couche particulier :

```
WITH unnested AS (
    SELECT
      item.awsaccountid AS account_id,
      item.awsregion AS region,
      item.configuration AS lambda_configuration,
      item.resourceid AS resourceid,
      item.resourcename AS resourcename,
      item.configuration AS configuration,
      json_parse(item.configuration) AS lambda_json
    FROM
      default.aws_config_configuration_snapshot,
      UNNEST(configurationitems) as t(item)
    WHERE
      "dt" = 'latest'
      AND item.resourcetype = 'AWS::Lambda::Function'
  )

  SELECT DISTINCT
    region as Region,
    resourcename as FunctionName,
    json_extract_scalar(lambda_json, '$.memorySize') AS memory_size,
    json_extract_scalar(lambda_json, '$.timeout') AS timeout,
    json_extract_scalar(lambda_json, '$.version') AS version
  FROM
    unnested
  WHERE
    lambda_configuration LIKE '%arn:aws:lambda:us-
east-1:111122223333:layer:AnyGovernanceLayer:24%'
```

Voici les résultats de la requête :

Query results | Query stats

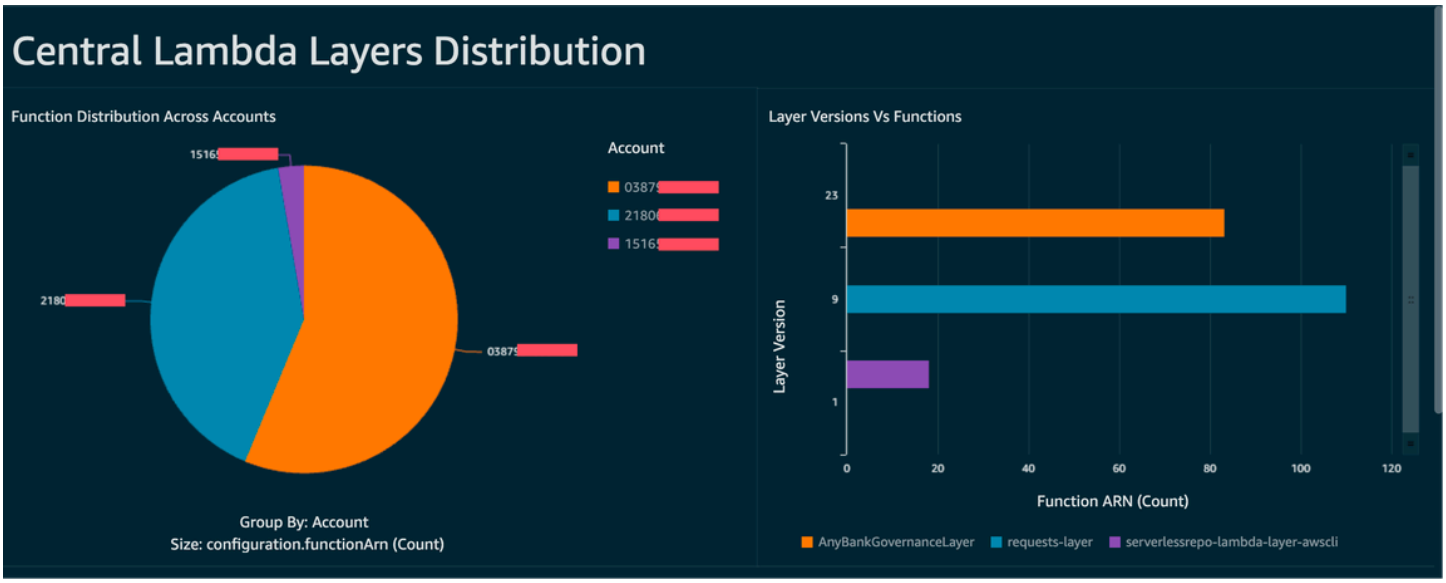
Completed — Time in queue: 127 ms — Run time: 1.803 sec — Data scanned: 239.40 KB

Results (27) — Copy — Download results

Search rows — 1

#	Region	FunctionName	memory_size	timeout	version
1	us-east-1	UpdateUIForPublishEvents	128	18	$LATEST
2	us-east-1	SchedulerCLI-InstanceSchedulerMain	128	300	$LATEST
3	us-east-1	my_functions_function10	128	3	$LATEST
4	us-east-1	lex-web-ui-CognitoIdentityP-CleanStackNameFunction-1TSORSH6L6YXQ	128	300	$LATEST
5	us-east-1	GetLatestArn	128	3	$LATEST
6	us-east-1	aws-python-http-api-project-dev-hello	1024	6	$LATEST
7	us-east-1	cloud9-MyTest-MyTest-688JGPVYP37L	128	15	$LATEST
8	us-east-1	my_functions_function1	128	3	$LATEST
9	us-east-1	my_functions_function25	128	3	$LATEST

Les données AWS Config étant agrégées au sein de l'organisation, vous pouvez ensuite créer un tableau de bord à l'aide d'Amazon Quick Suite. En important vos résultats Athena dans Amazon Quick Suite, vous pouvez visualiser dans quelle mesure vos fonctions Lambda respectent la règle de version des couches. Ce tableau de bord peut mettre en évidence les ressources conformes et non conformes, ce qui vous aide à déterminer votre politique d'application, comme indiqué dans la section suivante. L'image suivante est un exemple de tableau de bord qui indique la distribution des versions de couches appliquées aux fonctions au sein de l'organisation.



Phase 3 : mettre en œuvre et appliquer

Vous pouvez désormais éventuellement associer la règle de version de couche que vous avez créée lors de la phase 1 à une action de correction via un document d'automatisation de Systems Manager, que vous créez sous forme de script Python écrit avec AWS SDK pour Python (Boto3). Le script

appelle l'action d'API UpdateFunctionConfiguration pour chaque fonction Lambda, mettant à jour la configuration de la fonction avec le nouvel ARN de couche. Vous pouvez également demander au script de soumettre une pull request au référentiel de code pour mettre à jour l'ARN de la couche. De cette façon, les futurs déploiements de code sont également mis à jour avec le bon ARN de couche.

Signature de code Lambda avec AWS Signer

AWS Signer est un service de signature de code entièrement géré qui vous permet de valider votre code par le biais d'une signature numérique afin de confirmer que le code n'a pas été modifié et qu'il provient d'un diffuseur de publication. AWS Signer peut être utilisé conjointement avec AWS Lambda pour vérifier que les fonctions et les couches ne sont pas modifiées avant le déploiement dans vos environnements AWS. Cela protège votre organisation contre les acteurs malveillants susceptibles d'avoir obtenu des informations d'identification pour créer de nouvelles fonctions ou mettre à jour des fonctions existantes.

Pour configurer la signature de code pour vos fonctions Lambda, commencez par créer un compartiment S3 avec la gestion des versions activée. Ensuite, créez un profil de signature avec AWS Signer, spécifiez Lambda comme plate-forme, puis spécifiez une période de jours pendant laquelle le profil de signature est valide. Exemple :

```
Signer:
  Type: AWS::Signer::SigningProfile
  Properties:
    PlatformId: AWSLambda-SHA384-ECDSA
    SignatureValidityPeriod:
      Type: DAYS
      Value: !Ref pValidDays
```

Utilisez ensuite le profil de signature et créez une configuration de signature avec Lambda. Vous devez spécifier ce qu'il convient de faire lorsque la configuration de signature détecte un artefact qui ne correspond pas à la signature numérique attendue : avertir (mais autoriser le déploiement) ou appliquer (et bloquer le déploiement). L'exemple ci-dessous est configuré pour appliquer et bloquer les déploiements.

```
SigningConfig:
  Type: AWS::Lambda::CodeSigningConfig
  Properties:
    AllowedPublishers:
      SigningProfileVersionArns:
        - !GetAtt Signer.ProfileVersionArn
    CodeSigningPolicies:
      UntrustedArtifactOnDeployment: Enforce
```

AWS Signer est à présent configuré avec Lambda pour bloquer les déploiements non fiables. Supposons que vous avez terminé de coder une demande de fonctionnalité et que vous êtes

maintenant prêt à déployer la fonction. La première étape consiste à compresser le code avec les dépendances appropriées, puis à signer l'artefact à l'aide du profil de signature que vous avez créé. Vous pouvez le faire en téléchargeant l'artefact zip dans le compartiment S3, puis en lançant une tâche de signature.

```
aws signer start-signing-job \
--source 's3={bucketName=your-versioned-bucket,key=your-prefix/your-zip-
artifact.zip,version=QyaJ3c4qa50LXV.9VaZgXHlsGbvCXxpT}' \
--destination 's3={bucketName=your-versioned-bucket,prefix=your-prefix/}' \
--profile-name your-signer-id
```

Vous obtenez un résultat comme suit, où le `jobId` est l'objet créé dans le compartiment et le préfixe de destination et `jobOwner` l'identifiant Compte AWS à 12 chiffres où le travail a été exécuté.

```
{
    "jobId": "87a3522b-5c0b-4d7d-b4e0-4255a8e05388",
    "jobOwner": "111122223333"
  }
```

Vous pouvez désormais déployer votre fonction à l'aide de l'objet S3 signé et de la configuration de signature de code que vous avez créée.

```
  Fn:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: s3://your-versioned-bucket/your-prefix/87a3522b-5c0b-4d7d-
b4e0-4255a8e05388.zip
      Handler: fn.handler
      Role: !GetAtt FnRole.Arn
      CodeSigningConfigArn: !Ref pSigningConfigArn
```

Vous pouvez également tester le déploiement d'une fonction avec l'artefact zip source non signé d'origine. Le déploiement devrait échouer avec le message d'erreur suivant : .

```
Lambda cannot deploy the function. The function or layer might be signed using a
 signature that the client is not configured to accept. Check the provided signature
 for unsigned.
```

Si vous créez et déployez vos fonctions à l'aide de AWS Serverless Application Model (AWS SAM), la commande package gère le téléchargement de l'artefact zip vers S3, lance également le travail

de signature et obtient l'artefact signé. Pour ce faire, vous pouvez utiliser la commande avec les paramètres suivants :

```
sam package -t your-template.yaml \
--output-template-file your-output.yaml \
--s3-bucket your-versioned-bucket \
--s3-prefix your-prefix \
--signing-profiles your-signer-id
```

AWS Signer vous permet de vérifier que le déploiement des artefacts ZIP déployés dans vos comptes est fiable. Vous pouvez inclure le processus ci-dessus dans vos pipelines CI/CD et exiger que toutes les fonctions soient associées à une configuration de signature de code en utilisant les techniques décrites dans les rubriques précédentes. En utilisant la signature de code dans vos déploiements de fonctions Lambda, vous empêchez les acteurs malveillants susceptibles d'avoir obtenu des informations d'identification pour créer ou mettre à jour des fonctions d'injecter du code malveillant dans vos fonctions.

Automatisation des évaluations de sécurité pour Lambda avec Amazon Inspector

Amazon Inspector est un service de gestion des vulnérabilités qui analyse continuellement les charges de travail pour détecter les vulnérabilités logicielles connues et l'exposition involontaire au réseau. Amazon Inspector crée une constatation qui décrit la vulnérabilité, identifie la ressource affectée, évalue la gravité de la vulnérabilité et fournit des conseils pour y remédier.

Le support d'Amazon Inspector fournit une évaluation automatique et continue des vulnérabilités de sécurité pour les fonctions et les couches Lambda. Amazon Inspector propose deux types de scan pour Lambda :

- Analyse standard Lambda (par défaut) : analyse les dépendances des applications au sein d'une fonction Lambda et de ses couches pour détecter les vulnérabilités des packages.
- Analyse du code Lambda : analyse le code d'application personnalisé dans vos fonctions et couches pour détecter les vulnérabilités du code. Vous pouvez activer l'analyse standard Lambda ou activer l'analyse standard Lambda conjointement avec l'analyse du code Lambda.

Pour activer Amazon Inspector, accédez à la console Amazon Inspector, développez la section Paramètres et choisissez Gestion des comptes. Dans l'onglet Comptes, choisissez Activer, puis sélectionnez l'une des options d'analyse.

Vous pouvez activer Amazon Inspector pour plusieurs comptes et déléguer les autorisations de gestion d'Amazon Inspector pour l'organisation à des comptes spécifiques lors de la configuration d'Amazon Inspector. Lors de l'activation, vous devez accorder des autorisations à Amazon Inspector en créant le rôle :`AWSServiceRoleForAmazonInspector2`. La console Amazon Inspector vous permet de créer ce rôle à l'aide d'une option en un clic.

Pour l'analyse standard Lambda, Amazon Inspector lance des analyses de vulnérabilité des fonctions Lambda dans les situations suivantes :

- Dès qu'Amazon Inspector découvre une fonction Lambda existante.
- Lorsque vous déployez une fonction Lambda.
- Lorsque vous déployez une mise à jour du code d'application ou des dépendances d'une fonction Lambda existante ou de ses couches.
- Chaque fois qu'Amazon Inspector ajoute un nouvel élément Common Vulnerabilities and Exposures (CVE) à sa base de données, et que ce CVE est pertinent pour votre fonction.

Pour l’analyse du code Lambda, Amazon Inspector évalue le code d'application de votre fonction Lambda à l'aide d'un raisonnement automatisé et d'un machine learning qui analyse le code de votre application pour vérifier sa conformité globale en matière de sécurité. Si Amazon Inspector détecte une vulnérabilité dans le code d'application de votre fonction Lambda, Amazon Inspector produit une recherche détaillée de vulnérabilité dans le code. Pour obtenir la liste des détections possibles, consultez la bibliothèque de détecteurs Amazon CodeGuru.

Pour consulter les résultats, rendez-vous sur la console Amazon Inspector. Dans le menu Résultats, choisissez Par fonction Lambda pour afficher les résultats de l’analyse de sécurité effectuée sur les fonctions Lambda.

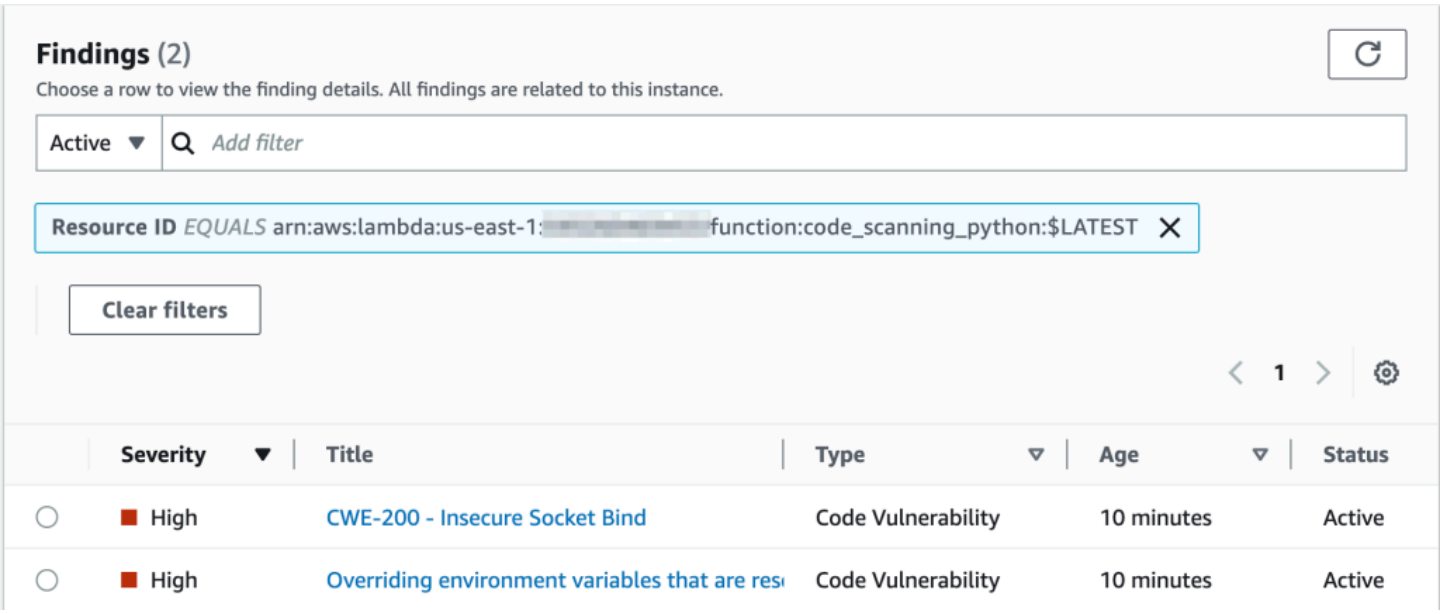
Pour exclure une fonction Lambda de l’analyse standard, balisez la fonction avec la paire clé-valeur suivante :

- `Key:InspectorExclusion`
- `Value:LambdaStandardScanning`

Pour exclure une fonction Lambda des analyses du code, balisez la fonction avec la paire clé-valeur suivante :

- `Key:InspectorCodeExclusion`
- `Value:LambdaCodeScanning`

Par exemple, comme le montre l'image suivante, Amazon Inspector détecte automatiquement les vulnérabilités et classe les résultats de type Vulnérabilité de code, ce qui indique que la vulnérabilité se trouve dans le code de la fonction, et non dans l'une des bibliothèques dépendantes du code. Vous pouvez vérifier ces informations pour une fonction spécifique ou pour plusieurs fonctions à la fois.



Vous pouvez approfondir chacun de ces résultats et découvrir comment remédier au problème.

Overriding environment variables that are reserved by AWS Lambda might lead to unexpected behavior.

Finding ID: arn:aws:inspector2:us-east-1:[redacted]:finding/[redacted]

Overriding environment variables that are reserved by AWS Lambda might lead to unexpected behavior or failure of the Lambda function.

Finding overview	
AWS account ID	[redacted]
Severity	High
Type	Code Vulnerability
Detector name	Override of reserved variable names in a Lambda function
Relevant CWE	--
Rule ID	Rule-434311
Detector tags	#availability, #aws-python-sdk, #aws-lambda, #data-integrity, #maintainability, #security, #security-context, #python
Fix available	Yes
Created at	March 29, 2023 10:08 AM (UTC-04:00)

Vulnerability details	
File path	lambda_function.py

Vulnerability location

```
3  import socket
4
5  def lambda_handler(event, context):
6
7      # print("Scenario 1");
8      os.environ['_HANDLER'] = 'hello'
9      # print("Scenario 1 ends")
10
11     # print("Scenario 2");
12     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
13     s.bind(('',0))
```

Suggested remediation

Your code attempts to override an environment variable that is reserved by the Lambda runtime environment. This can lead to unexpected behavior and might break the execution of your Lambda function.

Lorsque vous travaillez avec vos fonctions Lambda, assurez-vous de respecter les conventions de dénomination de vos fonctions Lambda. Pour de plus amples informations, consultez Utilisation des variables d'environnement Lambda.

Vous êtes responsable des suggestions de correction que vous acceptez. Passez toujours en revue les suggestions de correction avant de les accepter. Vous devrez peut-être apporter des modifications aux suggestions de correction pour vous assurer que votre code répond à vos attentes.

Mettre en œuvre l'observabilité pour la sécurité et la conformité Lambda

AWS Config est un outil utile pour rechercher et corriger les ressources AWS sans serveur non conformes. Chaque modification que vous apportez à vos ressources sans serveur est enregistrée dans AWS Config. Vous AWS Config permet également de stocker des données de capture instantanée de configuration sur S3. Vous pouvez utiliser Amazon Athena et Amazon Quick Suite pour créer des tableaux de bord et consulter les données. AWS Config Dans Détection des déploiements et des configurations Lambda non conformes avec AWS Config, nous avons discuté de la manière dont nous pouvons visualiser une certaine configuration, comme les couches Lambda. Cette rubrique développe ces concepts.

Visibilité sur les configurations Lambda

Vous pouvez utiliser des requêtes pour extraire des configurations importantes telles que l' Compte AWS ID, la région, la configuration de AWS X-Ray suivi, la configuration VPC, la taille de la mémoire, le temps d'exécution et les balises. Voici un exemple de requête que vous pouvez utiliser pour extraire ces informations d'Athena :

```
WITH unnested AS (
    SELECT
      item.awsaccountid AS account_id,
      item.awsregion AS region,
      item.configuration AS lambda_configuration,
      item.resourceid AS resourceid,
      item.resourcename AS resourcename,
      item.configuration AS configuration,
      json_parse(item.configuration) AS lambda_json
    FROM
      default.aws_config_configuration_snapshot,
      UNNEST(configurationitems) as t(item)
    WHERE
      "dt" = 'latest'
      AND item.resourcetype = 'AWS::Lambda::Function'
  )

  SELECT DISTINCT
    account_id,
    tags,
    region as Region,
    resourcename as FunctionName,
    json_extract_scalar(lambda_json, '$.memorySize') AS memory_size,
```

```
    json_extract_scalar(lambda_json, '$.timeout') AS timeout,
    json_extract_scalar(lambda_json, '$.runtime') AS version
    json_extract_scalar(lambda_json, '$.vpcConfig.SubnetIds') AS vpcConfig
    json_extract_scalar(lambda_json, '$.tracingConfig.mode') AS tracingConfig
  FROM
    unnested
```

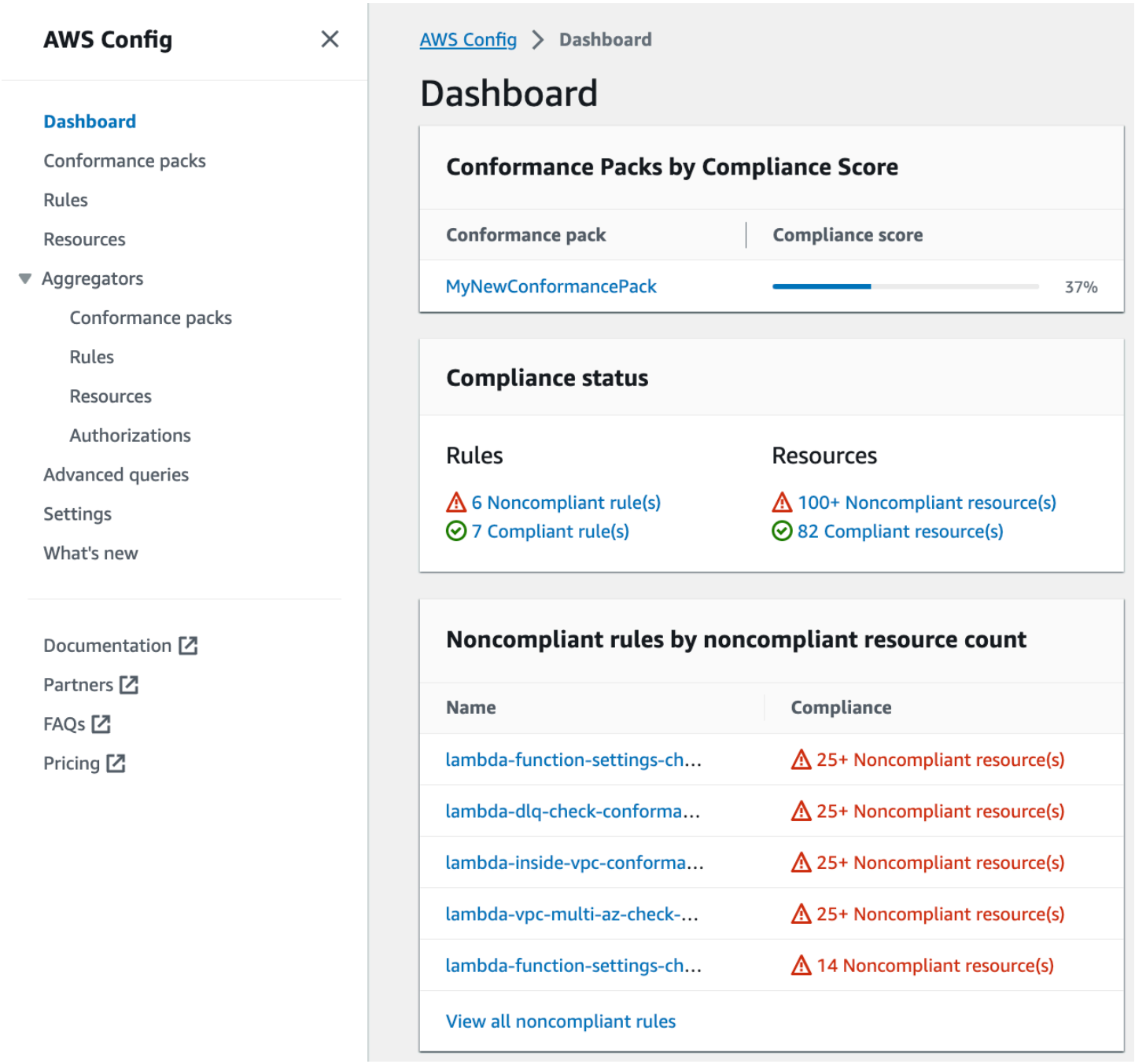
Vous pouvez utiliser la requête pour créer un tableau de bord Amazon Quick Suite et visualiser les données. Pour agréger les données de configuration des AWS ressources, créer des tables dans Athena et créer des tableaux de bord Quick Suite à partir des données d'Athena, consultez Visualisation des données à [l'aide d' AWS Config Athena et d'Amazon Quick Suite sur le blog Cloud Operations and] Management. AWS Notamment, cette requête récupère également les informations de balise pour les fonctions. Cela permet de mieux comprendre vos charges de travail et vos environnements, en particulier si vous utilisez des balises personnalisées.

Pour plus d'informations sur les actions que vous pouvez entreprendre, consultez la section [Prise en compte des constatations relatives à l'observabilité] plus loin dans cette rubrique.

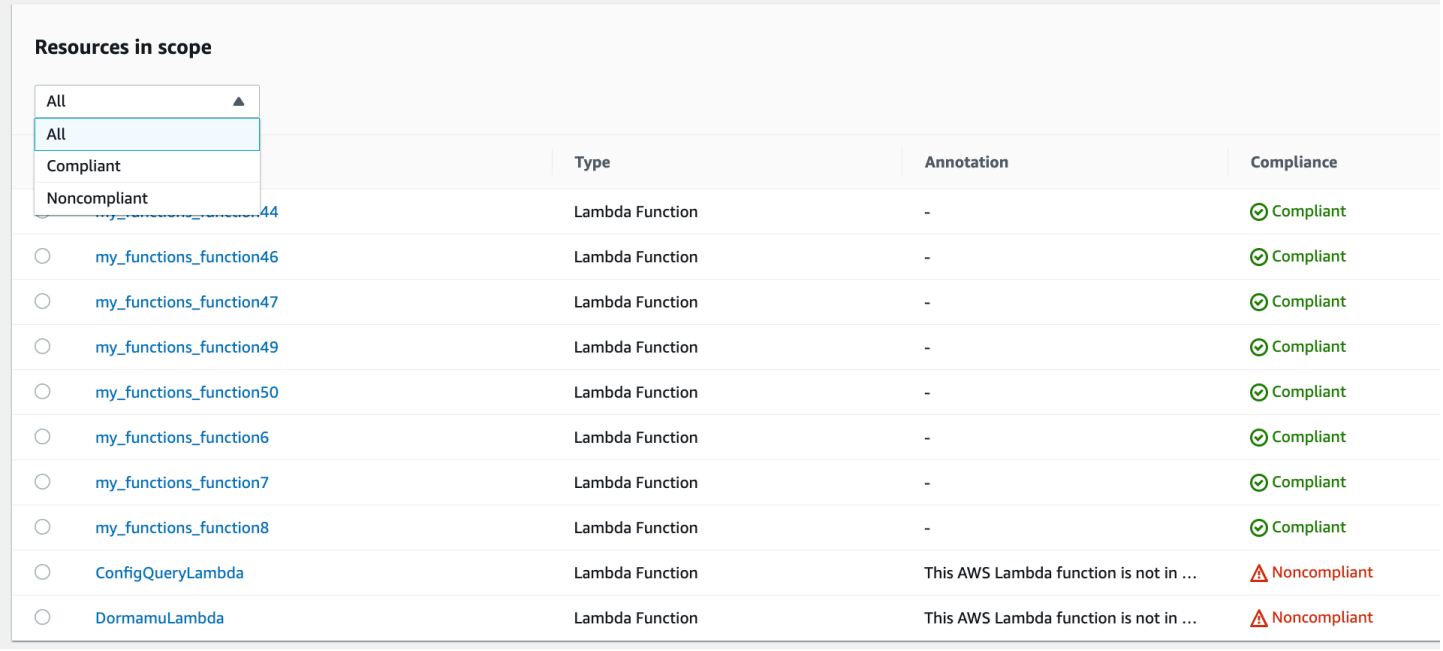
Visibilité sur la conformité à Lambda

Avec les données générées par AWS Config, vous pouvez créer des tableaux de bord au niveau de l'organisation pour surveiller la conformité. Cela permet un suivi et une surveillance cohérents de :

- Packs de conformité par score de conformité
- Règles en fonction des ressources non conformes
- Statut de conformité

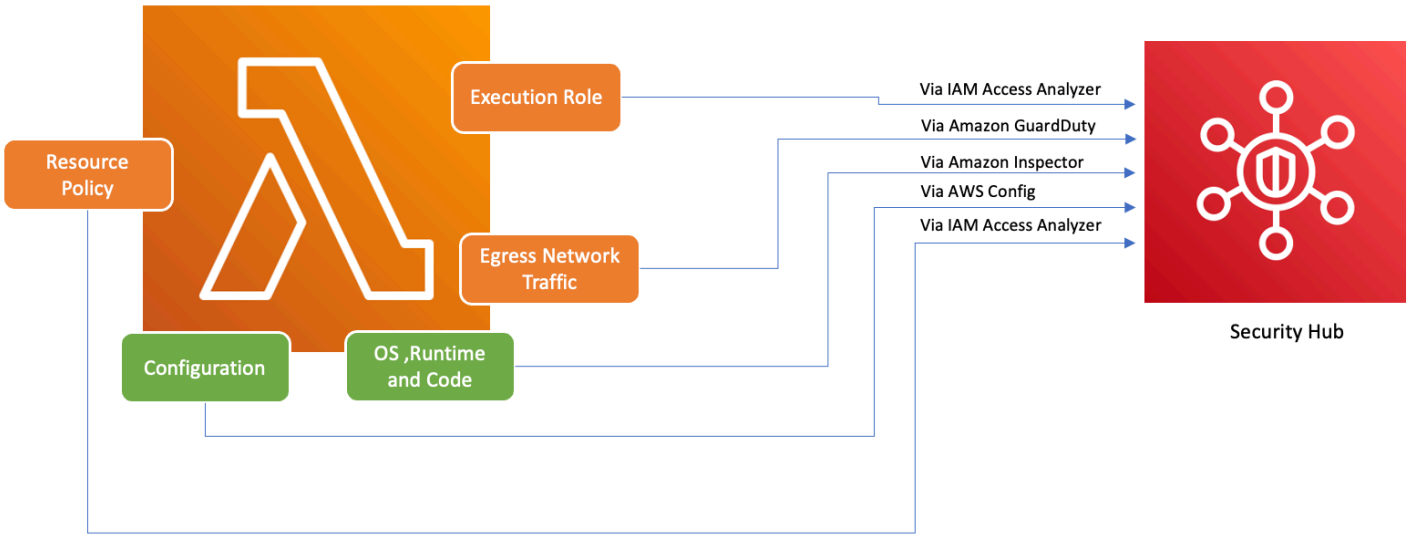


Vérifiez chaque règle pour identifier les ressources non conformes à cette règle. Par exemple, si votre organisation impose que toutes les fonctions Lambda soient associées à un VPC et si vous avez déployé une AWS Config règle pour identifier la conformité, vous pouvez sélectionner la `lambda-inside-vpc` règle dans la liste ci-dessus.



Pour plus d'informations sur les actions que vous pouvez entreprendre, consultez la section [Prise en compte des constatations relatives à l'observabilité] ci-dessous.

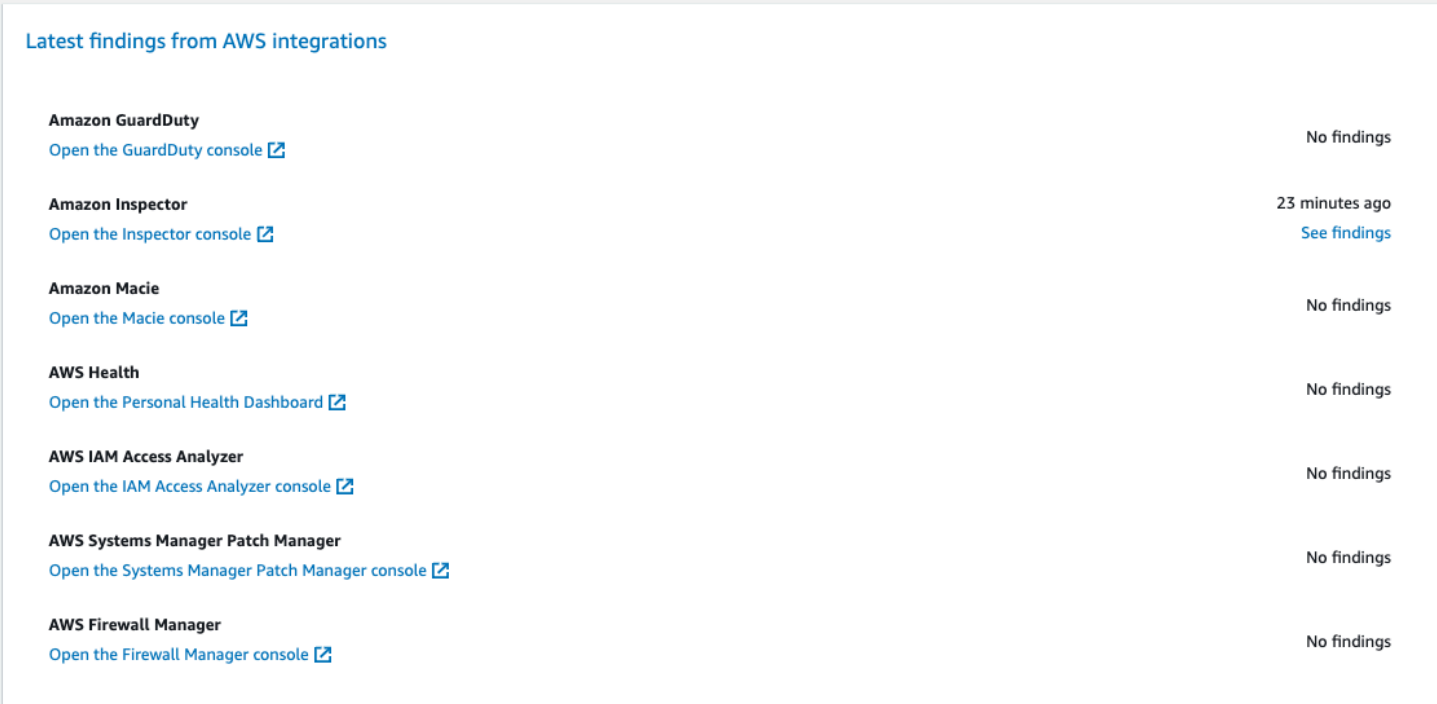
Visibilité des limites des fonctions Lambda à l'aide de Security Hub CSPM



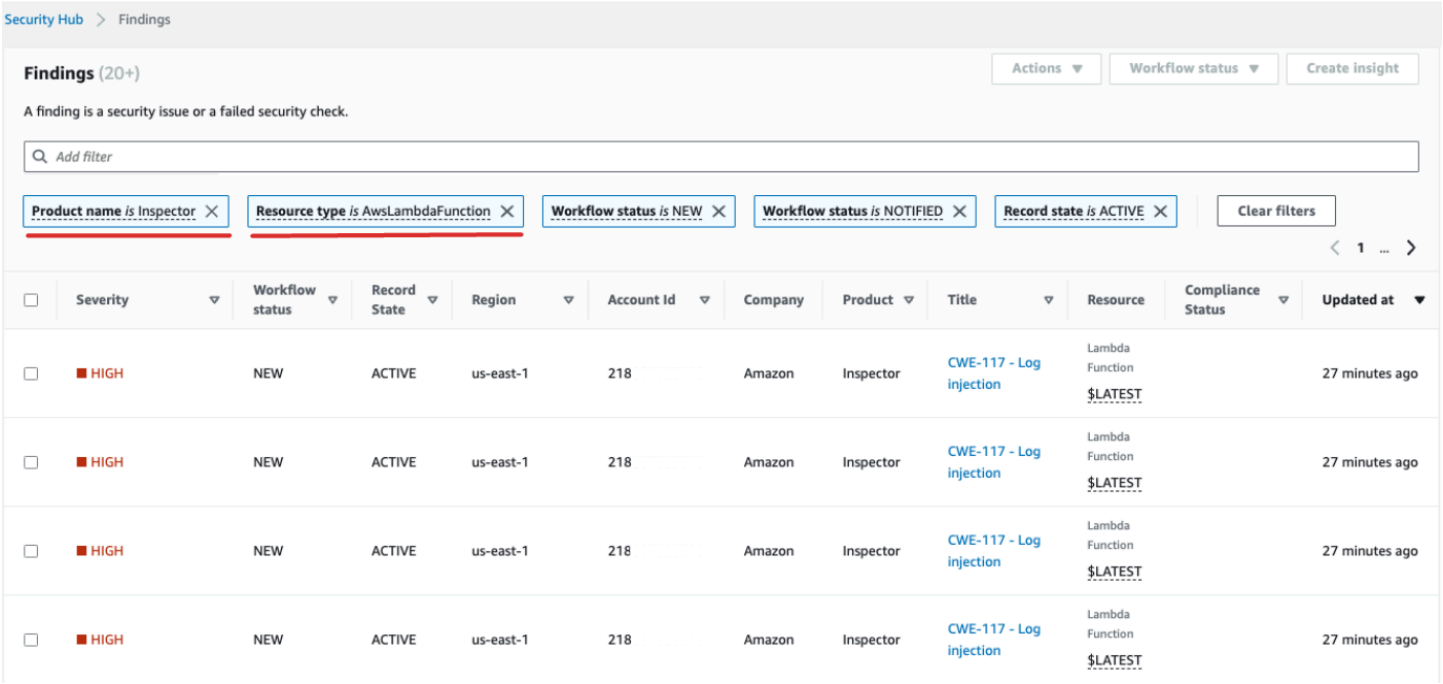
Pour garantir que les AWS services tels que Lambda sont utilisés en toute sécurité, nous avons AWS introduit les meilleures pratiques de sécurité fondamentales v1.0.0. Cet ensemble de bonnes pratiques fournit des directives claires pour sécuriser les ressources et les données dans l' AWS environnement, en soulignant l'importance de maintenir une posture de sécurité solide. Il AWS Security Hub CSPM complète cela en proposant un centre de sécurité et de conformité unifié. Il

regroupe, organise et hiérarchise les résultats de sécurité provenant de plusieurs AWS services tels qu'Amazon Inspector AWS Identity and Access Management Access Analyzer et Amazon. GuardDuty

Si Security Hub CSPM, Amazon Inspector, IAM Access Analyzer sont GuardDuty activés au sein de votre organisation, AWS Security Hub CSPM agrège automatiquement les résultats de ces services. Prenons par exemple Amazon Inspector. Security Hub CSPM vous permet d'identifier efficacement les vulnérabilités du code et des packages dans les fonctions Lambda. Dans la console Security Hub CSPM, accédez à la section inférieure intitulée Dernières découvertes en matière d' AWS intégrations. Ici, vous pouvez consulter et analyser les résultats provenant de divers AWS services intégrés.



Pour voir les détails, cliquez sur le lien Voir les résultats dans la deuxième colonne. Cela affiche une liste des résultats filtrés par produit, comme Amazon Inspector. Pour limiter votre recherche aux fonctions Lambda, définissez `ResourceType` sur `AwsLambdaFunction`. Cela affiche les résultats d'Amazon Inspector relatifs aux fonctions Lambda.



En GuardDuty effet, vous pouvez identifier les modèles de trafic réseau suspects. De telles anomalies peuvent suggérer l'existence d'un code potentiellement malveillant dans votre fonction Lambda.

Avec IAM Access Analyzer, vous pouvez vérifier les politiques, en particulier celles comportant des instructions de condition qui accordent l'accès aux fonctions à des entités externes. De plus, IAM Access Analyzer évalue les autorisations définies lors de l'utilisation de l'AddPermissionopération dans l'API Lambda avec un. `EventSourceToken`

Prise en compte des constatations relatives à l'observabilité

Compte tenu du large éventail de configurations possibles pour les fonctions Lambda et de leurs exigences distinctes, une solution d'automatisation standardisée pour la correction peut ne pas convenir à toutes les situations. De plus, les modifications sont mises en œuvre différemment selon les environnements. Si vous rencontrez une configuration qui ne semble pas conforme, tenez compte des directives suivantes :

1. Stratégie de balisage

   Nous vous recommandons de mettre en œuvre une stratégie de balisage complète. Chaque fonction Lambda doit être étiquetée avec des informations clés telles que :

   - Propriétaire : personne ou équipe responsable de la fonction.

- Environnement : production, mise en scène, développement ou bac à sable.
- Application : le contexte plus large auquel appartient cette fonction, le cas échéant.

2. Sensibilisation des propriétaires

   Au lieu d'automatiser les modifications majeures (comme l'ajustement de la configuration du VPC), contactez de manière proactive les propriétaires des fonctions non conformes (identifiées par le tag du propriétaire) en leur laissant suffisamment de temps pour :

   - Ajuster les configurations non conformes sur les fonctions Lambda.
   - Fournir une explication et demander une exception, ou affiner les normes de conformité.

3. Gestion d'une base de gestion des configurations (CMDB)

   Bien que les balises puissent fournir un contexte immédiat, le maintien d'une CMDB centralisée peut fournir des informations plus approfondies. Il peut contenir des informations plus détaillées sur chaque fonction Lambda, ses dépendances et d'autres métadonnées critiques. Une CMDB est une ressource inestimable pour les audits, les contrôles de conformité et l'identification des responsables des fonctions.

Le paysage de l'infrastructure sans serveur étant en constante évolution, il est essentiel d'adopter une position proactive en matière de surveillance. Grâce à des outils tels que AWS Config Security Hub CSPM et Amazon Inspector, les anomalies potentielles ou les configurations non conformes peuvent être rapidement identifiées. Cependant, les outils ne peuvent à eux seuls garantir une conformité totale ou des configurations optimales. Il est essentiel d'associer ces outils à des processus bien documentés et aux meilleures pratiques.

- Boucle de rétroaction : une fois les étapes de correction entreprises, assurez-vous qu'il existe une boucle de rétroaction. Cela implique de revoir régulièrement les ressources non conformes pour vérifier si elles ont été mises à jour ou si elles présentent toujours les mêmes problèmes.
- Documentation : documentez toujours les observations, les mesures prises et les exceptions accordées. Une documentation appropriée aide non seulement lors des audits, mais contribue également à améliorer le processus afin d'améliorer la conformité et la sécurité à l'avenir.
- Formation et sensibilisation : assurez-vous que toutes les parties prenantes, en particulier les responsables des fonctions Lambda, sont régulièrement formées et informées des meilleures pratiques, des politiques organisationnelles et des mandats de conformité. Des ateliers, des webinaires ou des sessions de formation réguliers peuvent contribuer dans une large mesure

à garantir que tout le monde est sur la même longueur d'onde en matière de sécurité et de conformité.

En conclusion, alors que les outils et les technologies fournissent des capacités robustes pour détecter et signaler les problèmes potentiels, l'élément humain (compréhension, communication, formation et documentation) reste essentiel. Ensemble, ils forment une puissante combinaison qui garantit que vos fonctions Lambda et votre infrastructure globale restent conformes, sécurisées et optimisées pour répondre aux besoins de votre entreprise.

Validation de la conformité pour AWS Lambda

Les auditeurs tiers évaluent la sécurité et la conformité de AWS Lambda dans le cadre de plusieurs programmes de conformité AWS. Il s'agit notamment des certifications SOC, PCI, FedRAMP, HIPAA et autres.

Pour obtenir la liste des services AWS concernés par des programmes de conformité spécifiques, consultez Services AWS concernés par les programmes de conformité. Pour obtenir des informations générales, veuillez consulter Programmes de conformité AWS.

Vous pouvez télécharger les rapports de l'audit externe avec AWS Artifact. Pour plus d'informations, consultez Téléchargement de rapports dans AWS Artifact.

Votre responsabilité de conformité lors de l'utilisation de Lambda est déterminée par la sensibilité de vos données, les objectifs de conformité de votre entreprise, ainsi que par la législation et la réglementation applicables. Vous pouvez mettre en œuvre des contrôles de gouvernance pour garantir que les fonctions Lambda de votre entreprise répondent à vos exigences de conformité. Pour de plus amples informations, consultez Création d’une stratégie de gouvernance pour les couches et les fonctions Lambda.

Résilience dans AWS Lambda

L’infrastructure mondiale d’AWS repose sur les Régions AWS et les zones de disponibilité AWS. Les régions fournissent plusieurs zones de disponibilité physiquement séparées et isolées, reliées par un réseau à latence faible, à haut débit et hautement redondant. Avec les zones de disponibilité, vous pouvez concevoir et exploiter des applications et des bases de données qui basculent automatiquement d’une zone de disponibilité à l’autre sans interruption. Les zones de disponibilité sont plus hautement disponibles, tolérantes aux pannes et évolutives que les infrastructures traditionnelles à un ou plusieurs centres de données.

Pour de plus amples informations sur les régions et les zones de disponibilité AWS, veuillez consulter Infrastructure mondiale AWS.

Outre l’infrastructure globale d’AWS, Lambda propose plusieurs fonctions qui contribuent à la prise en charge de vos besoins en matière de résilience et de sauvegarde des données.

- Gestion des versions – Vous pouvez utiliser la gestion des versions dans Lambda pour enregistrer le code et la configuration de votre fonction à mesure que vous la développez. Avec les alias, vous

pouvez utiliser la gestion des versions pour effectuer des déploiements bleu/vert et de roulement. Pour plus d'informations, consultez Gestion des versions d'une fonction Lambda.

- Mise à l'échelle – Lorsque votre fonction reçoit une demande tandis qu'elle traite une demande précédente, Lambda lance une autre instance de votre fonction pour gérer la charge accrue. Lambda adapte automatiquement son échelle pour gérer 1 000 exécutions simultanées par région, un quota qui peut être augmenté si nécessaire. Pour en savoir plus, consultez Présentation de la mise à l'échelle de fonction Lambda.
- Haute disponibilité – Lambda exécute votre fonction dans plusieurs zones de disponibilité afin de vous assurer qu'elle est disponible pour traiter les événements en cas d'interruption de service dans une seule zone. Si vous configurez votre fonction pour vous connecter à un cloud privé virtuel (VPC) de votre compte, spécifiez les sous-réseaux dans plusieurs zones de disponibilité, pour une garantie de haute disponibilité. Pour plus d'informations, consultez Octroi aux fonctions Lambda d'un accès aux ressources d'un Amazon VPC.
- Simultanéité réservée – Pour vous assurer que votre fonction peut toujours adapter son échelle afin de gérer des demandes supplémentaires, vous pouvez réserver de la simultanéité. La définition de la simultanéité réservée pour une fonction permet de s'assurer qu'elle peut être mise à l'échelle, sans le dépasser, un certain nombre spécifié d'appels simultanés. Cela garantit que vous ne perdez pas les demandes en raison d'autres fonctions utilisant toute la simultanéité disponible. Pour plus d'informations, consultez Configuration de la simultanéité réservée pour une fonction.
- Nouvelles tentatives – Pour les appels asynchrones et un sous-ensemble d'appels déclenchés par d'autres services, en cas d'erreur, Lambda effectue automatiquement de nouvelles tentatives espacées. Les autres clients et les Services AWS qui invoquent des fonctions de manière synchrone sont responsables de l'exécution des nouvelles tentatives. Pour en savoir plus, consultez Présentation du comportement des nouvelles tentatives dans Lambda.
- Files d'attente de lettres mortes – Pour les appels asynchrones, vous pouvez configurer Lambda de façon à envoyer à une file d'attente de lettres mortes les demandes qui ont échoué à toutes les tentatives. Une file d'attente de lettres mortes est une rubrique Amazon SNS ou une file d'attente Amazon SQS qui reçoit des événements en vue de leur dépannage ou d'un nouveau traitement. Pour en savoir plus, consultez Ajout d'une file d'attente de lettres mortes.

Sécurité de l'infrastructure dans AWS Lambda

En tant que service géré, AWS Lambda est protégé par les procédures de sécurité du réseau mondial AWS. Pour plus d'informations sur les services de sécurité AWS et la manière dont AWS protège l'infrastructure, consultez la section Sécurité du cloud AWS. Pour concevoir votre

environnement AWS en utilisant les meilleures pratiques en matière de sécurité de l'infrastructure, consultez la section Protection de l'infrastructure dans le Security Pillar AWS Well-Architected Framework (Pilier de sécurité de l'infrastructure Well-Architected Framework).

Vous utilisez les appels d'API publiés par AWS pour accéder à Lambda via le réseau. Les clients doivent prendre en charge les éléments suivants :

- Protocole TLS (Transport Layer Security). Nous exigeons TLS 1.2 et recommandons TLS 1.3.
- Ses suites de chiffrement PFS (Perfect Forward Secrecy) comme DHE (Ephemeral Diffie-Hellman) ou ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). La plupart des systèmes modernes tels que Java 7 et les versions ultérieures prennent en charge ces modes.

Sécurisation des charges de travail avec des points de terminaison publics

Pour les charges de travail accessibles au public, AWS fournit un certain nombre de fonctionnalités et de services qui peuvent aider à atténuer certains risques. Cette section aborde l'authentification et l'autorisation des utilisateurs de l'application ainsi que la protection des points de terminaison d'API.

Authentification et autorisation

L'authentification est liée à l'identité et l'autorisation concerne les actions. Utilisez l'authentification pour contrôler qui peut invoquer une fonction Lambda, puis utilisez l'autorisation pour contrôler qui peut faire quoi. Pour de nombreuses applications, IAM est suffisant pour gérer les deux mécanismes de contrôle.

Pour les applications utilisant des utilisateurs externes, telles que les applications Web ou mobiles, il est courant d'utiliser des jetons Web JSON (JWT) pour gérer l'authentification et l'autorisation. Contrairement à la gestion traditionnelle des mots de passe basée sur le serveur, les JWT sont transmis par le client à chaque demande. Ils constituent un moyen cryptographiquement sécurisé de vérifier l'identité et les demandes à l'aide des données transmises par le client. Pour les applications basées sur Lambda, cela vous permet de sécuriser chaque appel vers chaque point de terminaison d'API sans avoir recours à un serveur central pour l'authentification.

Vous pouvez implémenter des JWT avec Amazon Cognito, un service de répertoire d'utilisateurs capable de gérer l'enregistrement, l'authentification, la récupération des comptes et d'autres opérations courantes de gestion des comptes. Le cadre Amplify fournit des bibliothèques pour

simplifier l'intégration de ce service dans votre application frontale. Vous pouvez également envisager des services partenaires tiers comme Auth0.

Compte tenu du rôle critique d'un service de fournisseur d'identité en matière de sécurité, il est important d'utiliser des outils professionnels pour protéger votre application. Il n'est pas recommandé de créer vos propres services pour gérer l'authentification ou l'autorisation. Toute vulnérabilité dans les bibliothèques personnalisées peut avoir des répercussions importantes sur la sécurité de votre charge de travail et de ses données.

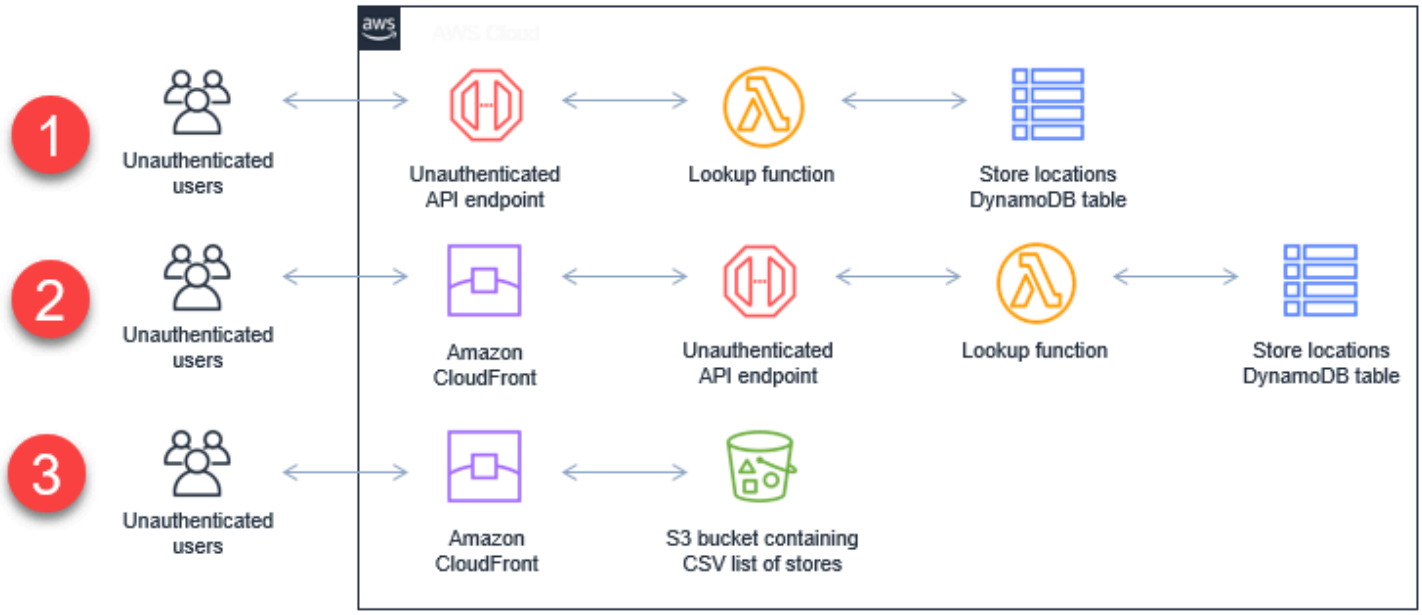
Protection des points de terminaison d'API

Pour les applications sans serveur, la méthode préférée pour servir publiquement une application dorsale consiste à utiliser Amazon API Gateway. Cela peut vous aider à protéger une API contre les utilisateurs malveillants ou les pics de trafic.

API Gateway propose deux types de points de terminaison pour les développeurs sans serveur : les API REST et les API HTTP. Les deux prennent en charge l'autorisation avec AWS Lambda, IAM ou Amazon Cognito. Lorsque vous utilisez IAM ou Amazon Cognito, les demandes entrantes sont évaluées et si elles ne contiennent pas le jeton requis ou si elles contiennent une authentification non valide, la demande est rejetée. Ces demandes ne vous sont pas facturées et elles ne sont pas prises en compte dans le calcul des quotas de limitation.

N'importe qui peut accéder aux routes d'API non authentifiées sur l'Internet public. Il est donc recommandé de limiter l'utilisation d'API non authentifiées. Si vous devez utiliser des API non authentifiées, il est important de les protéger contre les risques courants, tels que les attaques par déni de service (DoS). Appliquer AWS WAF à ces API peut aider à protéger votre application contre les attaques par injection SQL et par scripts inter-site (XSS). API Gateway implémente également la limitation au niveau du compte AWS et par client lorsque des clés d'API sont utilisées.

Dans de nombreux cas, les fonctionnalités fournies par une API non authentifiée peuvent être obtenues par une autre approche. Par exemple, une application Web peut fournir une liste des magasins de détail clients à partir d'une table DynamoDB aux utilisateurs non connectés. Cette demande peut provenir d'une application Web frontale ou de toute autre source qui appelle le point de terminaison de l'URL. Ce schéma compare trois solutions :



1. Cette API non authentifiée peut être appelée par quiconque sur Internet. Lors d'une attaque par déni de service, il est possible d'épuiser les limites de limitation des API, de simultanéité Lambda ou de capacité de lecture allouée par DynamoDB sur une table sous-jacente.
2. Une distribution CloudFront devant le point de terminaison de l'API avec une configuration de la durée de vie (TTL) appropriée absorberait la majeure partie du trafic lors d'une attaque DoS, sans modifier la solution sous-jacente pour récupérer les données.
3. Sinon, pour les données statiques qui changent rarement, la distribution CloudFront peut servir les données d'un compartiment Amazon S3.

Utilisation de la signature de code pour vérifier l'intégrité du code avec Lambda

La signature de code permet de s'assurer que seul du code approuvé est déployé dans vos fonctions Lambda. À l'aide de AWS Signer, vous pouvez créer des packages de code signés numériquement pour vos fonctions. Lorsque vous ajoutez une configuration de signature de code à une fonction, Lambda vérifie que tous les nouveaux déploiements de code sont signés par une source approuvée. Étant donné que les contrôles de validation de signature de code s'exécutent au moment du déploiement, l'exécution de la fonction n'est aucunement affectée.

> **Important**
>
> Les configurations de signature de code empêchent uniquement les nouveaux déploiements de code non signé. Si vous ajoutez une configuration de signature de code à une fonction existante contenant du code non signé, ce code continue de s'exécuter jusqu'à ce que vous déployiez un nouveau package de code.

Lorsque vous activez la signature de code pour une fonction, toutes les couches que vous ajoutez à la fonction doivent également être signées par un profil de signature autorisé.

Il n'y a aucun frais supplémentaire pour l'utilisation AWS Signer ou la signature de code pour AWS Lambda.

Validation de signature

Lambda effectue les contrôles de validation suivants lorsque vous déployez un package de code signé vers votre fonction :

1. Intégrité : vérifie que le package de code n'a pas été modifié depuis sa signature. Lambda compare le hachage du package au hachage de la signature.
2. Expiration : affirme que la signature du package de code n'a pas expiré.
3. Incompatibilité : affirme que le package de code est signé avec un profil de signature autorisé
4. Révocation : affirme que la signature du package de code n'a pas été révoquée.

Lorsque vous créez une configuration de signature de code, vous pouvez utiliser le UntrustedArtifactOnDeploymentparamètre pour spécifier la manière dont Lambda doit réagir en cas d'échec des contrôles d'expiration, d'incompatibilité ou de révocation. Vous pouvez choisir l'une de ces actions :

- `Warn` : il s'agit du paramètre par défaut. Lambda autorise le déploiement du package de code, mais émet un avertissement. Lambda émet une nouvelle CloudWatch métrique Amazon (`SignatureValidationErrors`) et enregistre également l'avertissement dans le CloudTrail journal.
- `Enforce` : Lambda émet un avertissement (identique à celui de l'action `Warn`) et bloque le déploiement du package de code.

Rubriques

- Création de configurations de signature de code pour Lambda
- Configuration des politiques IAM pour les configurations de signature de code Lambda
- Utilisation des balises dans les configurations de signature de code

Création de configurations de signature de code pour Lambda

Afin d'activer la signature de code pour une fonction, vous créez une configuration de signature de code et l'attachez à la fonction. Une configuration de signature de code définit une liste de profils de signature autorisés et l'action de stratégie à privilégier en cas d'échec de l'un des contrôles de validation.

> **Note**
>
> Les fonctions définies en tant qu'images de conteneur ne prennent pas en charge la signature de code.

Sections

- Conditions préalables à la configuration
- Création de configurations de signature de code
- Activation de la signature de code pour une fonction

Conditions préalables à la configuration

Avant de pouvoir configurer la signature de code pour une fonction Lambda, utilisez le Signataire AWS pour effectuer les opérations suivantes :

- Créer un ou plusieurs profils de signature.
- Utiliser un profil de signature afin de créer un package de code signé pour votre fonction.

Création de configurations de signature de code

Une configuration de signature de code définit une liste des profils de signature autorisés et la stratégie de validation de signature.

Pour créer une configuration de signature de code (console)

1. Ouvrez la page des configurations de signature de code de la console Lambda.
2. Choisissez Create configuration (Créer une configuration).
3. Pour Description, saisissez un nom descriptif pour la configuration.
4. Sous Signing profiles (Profils de signature), ajoutez jusqu'à 20 profils de signature à la configuration.
   a. Pour Signing profile version ARN (ARN de la version du profil de signature), sélectionnez l'Amazon Resource Name (ARN) de la version du profil, ou saisissez l'ARN.
   b. Pour ajouter un profil de signature supplémentaire, sélectionnez Add signing profiles (Ajouter des profils de signature).
5. Sous Signature validation policy (Stratégie de validation de signature), sélectionnez Warn (Avertissement) ou Enforce (Application).
6. Choisissez Create configuration (Créer une configuration).

Activation de la signature de code pour une fonction

Afin d'activer la signature de code pour une fonction, ajoutez une configuration de signature de code à la fonction.

> **Important**
>
> Les configurations de signature de code empêchent uniquement les nouveaux déploiements de code non signé. Si vous ajoutez une configuration de signature de code à une fonction existante contenant du code non signé, ce code continue de s'exécuter jusqu'à ce que vous déployiez un nouveau package de code.

Pour associer une configuration de signature de code à une fonction (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction pour laquelle vous souhaitez activer la signature de code.
3. Ouvrez l'onglet Configuration.
4. Faites défiler vers le bas et choisissez Signature de code.
5. Choisissez Modifier.

6. Sous Edit code signing (Modifier la signature de code), sélectionnez une configuration de signature de code pour cette fonction.
7. Choisissez Enregistrer.

Configuration des politiques IAM pour les configurations de signature de code Lambda

Pour accorder à un utilisateur l'autorisation d'accéder aux opérations d'API de signature de code Lambda, attachez une ou plusieurs instructions de stratégie à la stratégie utilisateur. Pour plus d'informations sur les stratégies utilisateur, consultez Politiques IAM basées sur l'identité pour Lambda.

L'exemple d'instruction de stratégie suivant autorise la création, la mise à jour et la récupération de configurations de signature de code.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
          "lambda:CreateCodeSigningConfig",
          "lambda:UpdateCodeSigningConfig",
          "lambda:GetCodeSigningConfig"
        ],
      "Resource": "*"
    }
  ]
}
```

Les administrateurs peuvent utiliser la clé de condition `CodeSigningConfigArn` pour spécifier les configurations de signature de code que les développeurs doivent utiliser pour créer ou mettre à jour vos fonctions.

L'exemple de déclaration de stratégie suivant accorde l'autorisation de créer une fonction. La déclaration de stratégie comprend une condition `lambda:CodeSigningConfigArn` pour spécifier

la configuration de signature de code autorisée. Lambda bloque les demandes `CreateFunction` d'API si le CodeSigningConfigArnparamètre est absent ou ne correspond pas à la valeur de la condition.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowReferencingCodeSigningConfig",
      "Effect": "Allow",
      "Action": [
        "lambda:CreateFunction"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "lambda:CodeSigningConfigArn": "arn:aws:lambda:us-
east-1:111122223333:code-signing-config:csc-0d4518bd353a0a7c6"
        }
      }
    }
  ]
}
```

Utilisation des balises dans les configurations de signature de code

Vous pouvez étiqueter les configurations de signature de code pour organiser et gérer vos ressources. Les balises sont des paires clé-valeur de forme libre associées à vos ressources prises en charge par l'ensemble des ressources Services AWS. Pour plus d'informations sur les cas d'utilisation des balises, consultez la section Stratégies de balisage courantes dans le guide des AWS ressources de balisage et de l'éditeur de balises.

Vous pouvez utiliser l' AWS Lambda API pour afficher et mettre à jour les balises. Vous pouvez également afficher et mettre à jour les balises tout en gérant une configuration de signature de code spécifique dans la console Lambda.

Sections

- Autorisations requises pour l’utilisation des balises
- Utilisation des balises avec la console Lambda
- Utilisation de balises avec le AWS CLI

Autorisations requises pour l’utilisation des balises

Pour autoriser une identité Gestion des identités et des accès AWS (IAM) – utilisateur, groupe ou rôle – à afficher ou marquer les ressources, accordez-lui les autorisations correspondantes :

- lambda : ListTags —Lorsqu' une ressource possède des balises, accordez cette autorisation à tous ceux qui ont besoin de `ListTags` l'utiliser. Pour les fonctions balisées, cette autorisation est également nécessaire pour `GetFunction`.
- lambda : TagResource —Accordez cette autorisation à toute personne ayant besoin d'appeler `TagResource` ou d'exécuter un tag lors de la création.

Vous pouvez éventuellement envisager d'accorder également l'UntagResourceautorisation lambda : pour autoriser les `UntagResource` appels à la ressource.

Pour de plus amples informations, veuillez consulter Politiques IAM basées sur l’identité pour Lambda.

Utilisation des balises avec la console Lambda

Vous pouvez utiliser la console Lambda pour créer des configurations de signature de code qui comportent des balises, pour ajouter des balises aux configurations de signature de code existantes et pour filtrer des configurations de signature de code par balise.

Ajout d’une balise lors de la création d’une configuration de signature de code

1. Ouvrez la page Configurations de signature de code dans la console Lambda.
2. Dans l’en-tête du volet de contenu, choisissez Créer une configuration.
3. Dans la section située en haut du volet de contenu, configurez votre signature de code. Pour plus d’informations sur la configuration des configurations de signature de code, consultez the section called “Signature de code”.
4. Dans la section Balises, choisissez Ajouter une balise.

5. Dans le champ Clé, saisissez la clé de votre balise. Pour plus d'informations sur les restrictions relatives au balisage, consultez la section Limites et exigences relatives au nommage des balises dans le Guide des AWS ressources de balisage et de l'éditeur de balises.
6. Choisissez Créer une configuration.

Pour ajouter une balise à une configuration de signature de code existante

1. Ouvrez la page Configurations de signature de code dans la console Lambda.
2. Sélectionnez le nom de votre configuration de signature de code.
3. Dans les onglets situés sous le volet Détails, sélectionnez Balises.
4. Choisissez Gérer les balises.
5. Choisissez Add new tag (Ajouter une nouvelle balise).
6. Dans le champ Clé, saisissez la clé de votre balise. Pour plus d'informations sur les restrictions relatives au balisage, consultez la section Limites et exigences relatives au nommage des balises dans le Guide des AWS ressources de balisage et de l'éditeur de balises.
7. Choisissez Enregistrer.

Pour filtrer les configurations de signature de code par balise

1. Ouvrez la page Configurations de signature de code dans la console Lambda.
2. Cliquez sur la barre de recherche.
3. Dans la liste déroulante, sélectionnez votre balise sous le sous-titre Balises.
4. Sélectionnez Utiliser : « tag-name » pour afficher toutes les configurations de signature de code étiquetées avec cette touche, ou choisissez un Opérateur pour affiner le filtrage en fonction de la valeur.
5. Sélectionnez votre valeur de balise pour appliquer un filtre combinant la clé et la valeur de la balise.

La zone de recherche prend également en charge la recherche de clés de balise. Saisissez le nom d’une clé pour la rechercher dans la liste.

Utilisation de balises avec le AWS CLI

Vous pouvez ajouter et supprimer des balises sur les ressources Lambda existantes, configurations de signature de code incluses, avec l’API Lambda. Il est également possible d’ajouter des balises

lors de la création d'une configuration de signature de code, vous permettant ainsi de conserver une ressource étiquetée tout au long de son cycle de vie.

Mise à jour des balises avec la balise Lambda APIs

Vous pouvez ajouter et supprimer des balises pour les ressources Lambda prises en charge par le biais des opérations TagResourceet de l'UntagResourceAPI.

Vous pouvez appeler ces opérations par l'intermédiaire de l' AWS CLI. Pour ajouter des balises à une ressource existante, utilisez la commande `tag-resource`. Cet exemple ajoute deux balises, l'une avec la clé *`Department`* et l'autre avec la clé*`CostCenter`*.

```
aws lambda tag-resource \
--resource arn:aws:lambda:us-east-2:123456789012:resource-type:my-resource \
--tags Department=Marketing,CostCenter=1234ABCD
```

Pour supprimer des balises, utilisez la commande `untag-resource`. Cet exemple supprime la balise contenant la clé*`Department`*.

```
aws lambda untag-resource --resource arn:aws:lambda:us-east-1:123456789012:resource-
type:resource-identifier \
--tag-keys Department
```

Ajout de balises lors de la création d'une configuration de signature de code

Pour créer une nouvelle configuration de signature de code Lambda avec des balises, utilisez l'opération CreateCodeSigningConfigAPI. Spécifiez le paramètre `Tags`. Vous pouvez appeler cette opération à l'aide de la `create-code-signing-config` AWS CLI commande et de l'`--tags`option. Pour plus d'informations sur la commande CLI, consultez create-code-signing-configla référence des AWS CLI commandes.

Avant d'utiliser le paramètre `Tags` avec `CreateCodeSigningConfig`, vérifiez que votre rôle dispose de l'autorisation d'étiqueter les ressources en plus des autorisations habituelles nécessaires à cette opération. Pour plus d'informations sur les autorisations requises pour l'étiquetage, consultez the section called "Autorisations requises pour l'utilisation des balises".

Afficher les tags avec le tag Lambda APIs

Pour afficher les balises associées à une ressource Lambda spécifique, utilisez l'opération d'API `ListTags`. Pour de plus amples informations, veuillez consulter ListTags.

Vous pouvez appeler cette opération à l'aide de la `list-tags` AWS CLI commande en fournissant un ARN (Amazon Resource Name).

```
aws lambda list-tags --resource arn:aws:lambda:us-east-1:123456789012:resource-type:resource-identifier
```

Filtrage de ressources par balise

Vous pouvez utiliser l'opération AWS Resource Groups Tagging API GetResourcesAPI pour filtrer vos ressources par balises. L'opération `GetResources` reçoit jusqu'à 10 filtres, chaque filtre contenant une clé de balise et jusqu'à 10 valeurs de balise. Vous fournissez `GetResources`avec un `ResourceType` pour filtrer par certains types de ressources.

Vous pouvez appeler cette opération à l'aide de la `get-resources` AWS CLI commande. Pour des exemples d'utilisation de `get-resources`, consultez get-resources dans la Référence des commandes de l'AWS CLI.

Surveillance, débogage et dépannage des fonctions Lambda

AWS Lambda s'intègre avec d'autres Services AWS pour vous aider à surveiller, suivre, déboguer et dépanner vos fonctions Lambda. Lambda surveille automatiquement les fonctions Lambda pour vous, et rapporte les métriques via Amazon CloudWatch. Pour vous aider à surveiller votre code lors de l'exécution de celui-ci, Lambda suit automatiquement le nombre de demandes, la durée d'invocation par demande et le nombre de demandes générant une erreur.

Vous pouvez utiliser d'autres Services AWS pour dépanner vos fonctions Lambda. Cette section explique comment utiliser ces Services AWS pour surveiller, suivre, déboguer et dépanner vos fonctions et applications Lambda. Pour plus de détails sur la journalisation des fonctions et les erreurs dans chaque environnement d'exécution, consultez les sections relatives à chaque environnement d'exécution.

Sections

- Tarification
- Utilisation des métriques CloudWatch avec Lambda
- Utilisation des journaux de fonction Lambda
- Journalisation des appels d'API AWS Lambda avec AWS CloudTrail
- Visualisez les invocations de fonctions Lambda à l'aide de AWS X-Ray
- Surveillance des performances de fonctions avec Amazon CloudWatch Lambda Insights
- Surveillance des applications Lambda
- Surveillance des performances d'applications avec la vigie applicative Amazon CloudWatch
- Déboguer à distance des fonctions Lambda avec Visual Studio Code

Tarification

CloudWatch propose une offre gratuite perpétuelle. Au-delà du seuil de cette offre gratuite, CloudWatch facture les métriques, les tableaux de bord, les alarmes, les journaux et les informations. Pour plus d'informations, consultez Tarification Amazon CloudWatch.

Utilisation des métriques CloudWatch avec Lambda

Lorsque votre fonction AWS Lambda termine le traitement d'un événement, Lambda envoie des métriques d'invocation à Amazon CloudWatch. Vous n'avez pas besoin d'octroyer des autorisations supplémentaires à votre rôle d'exécution pour recevoir des métriques de fonction, et il n'y a pas de frais supplémentaires pour ces métriques.

De nombreux types de métriques sont associés aux fonctions Lambda. Il s'agit notamment des métriques d'invocation, des métriques de performance, des métriques de simultanéité, des métriques d'invocation asynchrone et des métriques de mappage des sources d'événements. Pour de plus amples informations, consultez the section called "Types de métriques".

Dans la console CloudWatch, vous pouvez afficher ces métriques et créer des graphiques et des tableaux de bord à partir de celles-ci. Il est également possible de définir des alarmes pour réagir aux changements dans l'utilisation, les performances ou les taux d'erreur. Lambda envoie les données des métriques à CloudWatch à intervalles d'une minute. Si vous souhaitez obtenir des informations plus immédiates sur votre fonction Lambda, vous pouvez créer de métriques personnalisées haute définition. Des frais s'appliquent aux métriques personnalisées et aux alarmes CloudWatch. Pour en savoir plus, consultez Tarification Amazon CloudWatch.

Affichage des métriques pour une fonction Lambda

Utilisez la console CloudWatch pour consulter les métriques de vos fonctions Lambda. Dans la console, vous pouvez filtrer et trier les métriques des fonctions par nom de fonction, alias, version ou UUID de mappage des sources d'événements.

Pour afficher des métriques dans la console CloudWatch

1. Ouvrez la page Metrics (Métriques) (espace de noms `AWS/Lambda`) dans la console CloudWatch.
2. Dans l'onglet Parcourir, sous Métriques, choisissez l'une des dimensions suivantes :

   - By Function Name (Par nom de fonction) (`FunctionName`) – Affichez les métriques agrégées pour l'ensemble des versions et alias d'une fonction.
   - By Resource (Par ressource) (`Resource`) – Affichez les métriques pour une version ou un alias d'une fonction.
   - By Executed Version (Par version exécutée) (`ExecutedVersion`) – Affichez les métriques pour une combinaison d'alias et de version. Utilisez la dimension `ExecutedVersion` pour

comparer les taux d'erreur pour deux versions d'une fonction qui sont les cibles d'un alias pondéré.

- Par UUID de mappage des sources d'événement (`EventSourceMappingUUID`) : affiche les métriques d'un mappage des sources d'événement.
- Sur toutes les fonctions (aucun) : affichez les métriques agrégées pour toutes les fonctions dans la Région AWS actuelle.

3. Choisissez une métrique. La métrique doit apparaître automatiquement dans le graphique visuel, ainsi que sous l'onglet Graphiques des métriques.

Par défaut, les graphiques utilisent la statistique `Sum` pour toutes les métriques. Pour choisir une autre statistique et personnaliser le graphique, utilisez les options de l'onglet Graphique des métriques.

> **Note**
>
> L'horodatage d'une métrique indique le moment où la fonction a été invoquée. En fonction de la durée de l'invocation, il peut se rapporter à plusieurs minutes avant l'émission de la métrique. Par exemple, si le délai d'expiration de votre fonction est de 10 minutes, effectuez une recherche plus de 10 minutes dans le passé pour obtenir des métriques précises.

Pour de plus amples informations sur Amazon CloudWatch, consultez le Guide de l'utilisateur Amazon CloudWatch.

Types de métriques pour les fonctions Lambda

Cette section décrit les types de métriques Lambda disponibles dans la console CloudWatch.

Rubriques

- Métriques d'invocation
- Métriques de déploiement
- Métriques de performances
- Métriques de simultanéité
- Métriques d'invocations asynchrones
- Métriques de mappage des sources d'événements

Métriques d'invocation

Les métriques d'invocation sont des indicateurs binaires du résultat d'une invocation de fonction Lambda. Vous devez visualiser les métriques suivantes avec la statistique `Sum`. Par exemple, si la fonction renvoie une erreur, Lambda envoie la métrique `Errors` avec une valeur de 1. Pour obtenir le nombre d'erreurs de fonction qui se sont produites chaque minute, examinez la somme `Sum` de la métrique `Errors` avec une période d'une minute.

- `Invocations` – Nombre de fois où votre code de fonction est invoqué, y compris les invocations réussies et les invocations qui entraînent une erreur de fonction. Les invocations ne sont pas enregistrés si la demande d'invocation est limitée ou si elle entraîne une erreur d'invocation. La valeur de `Invocations` est égale au nombre de demandes facturées.
- `Errors` – Nombre d'invocations entraînant une erreur de fonction. Les erreurs de fonction incluent les exceptions levées par votre code et par l'exécution Lambda. L'environnement d'exécution renvoie des erreurs pour des problèmes tels que les expirations de délai et les erreurs de configuration. Pour calculer le taux d'erreur, divisez la valeur `Errors` par la valeur `Invocations`. Notez que l'horodatage d'une métrique d'erreur reflète l'heure d'invocation de la fonction, et non l'heure à laquelle l'erreur s'est produite.
- `DeadLetterErrors` : pour un invocation asynchrone, le nombre de fois où Lambda tente sans succès d'envoyer un événement à une file d'attente de lettres mortes (DLQ). Des erreurs de type « lettre morte » peuvent se produire en raison d'une définition incorrecte des ressources ou des limites de taille.
- `DestinationDeliveryFailures` – Pour l'invocation asynchrone et les mappages des sources d'événements pris en charge, le nombre de fois où Lambda tente d'envoyer un événement à une destination, mais échoue. Pour les mappages des sources d'événements, Lambda prend en charge les destinations des sources de flux (DynamoDB et Kinesis). Des erreurs de remise peuvent se produire en raison d'erreurs d'autorisations, de ressources mal configurées ou des limites de taille. Des erreurs peuvent également se produire si la destination que vous avez configurée est un type non pris en charge, tel qu'une file d'attente FIFO Amazon SQS ou une rubrique FIFO Amazon SNS.
- `Throttles` – Nombre de demandes d'invocation limitées. Lorsque toutes les instances de fonction traitent des requêtes et qu'aucune simultanéité n'est disponible pour effectuer une augmentation, Lambda rejette les requêtes supplémentaires avec une erreur `TooManyRequestsException`. Les demandes limitées et les autres erreurs d'invocation ne comptent pas comme `Invocations` ou `Errors`.

- `OversizedRecordCount` – Pour les sources d'événements Amazon DocumentDB, le nombre d'événements que votre fonction reçoit de votre flux de modifications et dont la taille est supérieure à 6 Mo. Lambda supprime le message et émet cette métrique.
- `ProvisionedConcurrencyInvocations` : le nombre d'invocation de votre code de fonction à l'aide de la simultanéité provisionnée.
- `ProvisionedConcurrencySpilloverInvocations` : le nombre d'invocations de votre code de fonction à l'aide d'une simultanéité standard lorsque toutes les simultanéités provisionnées sont utilisées.
- `RecursiveInvocationsDropped` : le nombre de fois où Lambda a arrêté l'invocation de votre fonction parce qu'il a détecté que votre fonction faisait partie d'une boucle récursive infinie. La détection des boucles récursives surveille le nombre de fois qu'une fonction est invoquée dans le cadre d'une chaîne de requêtes en suivant les métadonnées ajoutées par les kits SDK AWS compatibles. Si votre fonction est invoquée dans le cadre d'une chaîne de requêtes environ plus de 16 fois, Lambda abandonne l'invocation suivante. Si vous désactivez la détection des boucles récursives, cette métrique n'est pas émise. Pour en savoir plus sur cette fonctionnalité, consultez Utilisation de la détection de boucle récursive Lambda pour prévenir les boucles infinies.

Métriques de déploiement

Les métriques de déploiement fournissent des informations sur les événements de déploiement de la fonction Lambda et les processus de validation associés.

- `SignatureValidationErrors` : le nombre de fois qu'un déploiement de package de code s'est produit avec des échecs de validation de signature lorsque la politique de configuration de signature de code est définie sur `Warn`. Cette métrique est émise lorsque les vérifications d'expiration, d'incompatibilité ou de révocation échouent, mais que le déploiement est toujours autorisé en raison du paramètre de politique `Warn`. Pour plus d'informations sur la signature de code, consultez Utilisation de la signature de code pour vérifier l'intégrité du code avec Lambda.

Métriques de performances

Les métriques de performance fournissent des détails sur les performances d'un invocation de fonction individuel. Par exemple, la métrique `Duration` indique la durée en millisecondes que votre fonction consacre au traitement d'un événement. Pour avoir une idée de la rapidité avec laquelle votre fonction traite les événements, affichez ces métriques avec la statistique `Average` ou `Max`.

- `Duration` – Durée de traitement d'un événement par votre code de fonction. La durée facturée pour une invocation est la valeur de la `Duration` arrondie à la milliseconde la plus proche. La `Duration` n'inclut pas le temps de démarrage à froid.
- `PostRuntimeExtensionsDuration` – Durée cumulée que l'environnement d'exécution a consacrée à l'exécution de code pour des extensions une fois l'exécution du code de la fonction terminée.
- `IteratorAge` : pour les sources d'événements DynamoDB, Kinesis et Amazon DocumentDB, l'âge en millisecondes du dernier enregistrement dans l'événement. Ce métrique mesure le temps écoulé entre le moment où un flux reçoit l'enregistrement et le moment où le mappage des sources d'événements envoie l'événement à la fonction.
- `OffsetLag` : pour les sources d'événements autogérées Apache Kafka et Amazon Managed Streaming for Apache Kafka (Amazon MSK), la différence de décalage entre le dernier enregistrement écrit sur une rubrique et le dernier enregistrement traité par le groupe de consommateurs de votre fonction. Bien qu'une rubrique Kafka puisse comporter plusieurs partitions, cette métrique mesure le décalage de décalage au niveau de la rubrique.

`Duration` prend également en charge les statistiques de centiles (p). Utilisez les centiles pour exclure les valeurs aberrantes qui faussent les statistiques `Average` et `Maximum`. Par exemple, la statistique p95 indique la durée maximale de 95 % des invocations, en excluant les 5 % les plus lentes. Pour plus d'informations, veuillez consulter la rubrique Centiles dans le Guide de l'utilisateur Amazon CloudWatch.

Métriques de simultanéité

Lambda signale les métriques de simultanéité sous la forme d'une valeur agrégée du nombre d'instances traitant des événements au sein d'une fonction, d'une version, d'un alias ou d'une Région AWS. Pour voir la distance qui vous sépare des limites de simultanéité, affichez ces métriques à l'aide de la statistique `Max`.

- `ConcurrentExecutions` – Nombre d'instances de fonction qui traitent des événements. Si ce nombre atteint votre quota d'exécutions simultanées pour la région ou la limite de simultanéité réservée sur la fonction, Lambda limite les demandes d'invocation supplémentaires.
- `ProvisionedConcurrentExecutions` : le nombre d'instances de fonction qui traitent des événements à l'aide de la simultanéité provisionnée. Pour chaque invocation d'alias ou de version avec une simultanéité approvisionnée, Lambda émet le nombre actuel. Si votre fonction est inactive ou ne reçoit pas de requêtes, Lambda n'émet pas cette métrique.

- `ProvisionedConcurrencyUtilization` : pour une version ou un alias, la valeur de `ProvisionedConcurrentExecutions` divisée par la quantité totale de simultanéité allouée configurée. Par exemple, si vous configurez une simultanéité allouée de 10 pour votre fonction et que votre `ProvisionedConcurrentExecutions` est 7, alors votre `ProvisionedConcurrencyUtilization` est 0,7.

  Si votre fonction est inactive ou ne reçoit pas de requêtes, Lambda n'émet pas cette métrique, car elle est basée sur `ProvisionedConcurrentExecutions`. Gardez cela à l'esprit si vous l'utilisez `ProvisionedConcurrencyUtilization` comme base pour les alarmes CloudWatch.
- `UnreservedConcurrentExecutions` – Pour une région, nombre d'événements traités par des fonctions qui n'ont pas de simultanéité réservée.
- `ClaimedAccountConcurrency` – Pour une région, le niveau de simultanéité qui n'est pas disponible pour les appels à la demande. `ClaimedAccountConcurrency` est égal à `UnreservedConcurrentExecutions` plus le montant de la simultanéité allouée (c'est-à-dire le total de la simultanéité réservée plus le total de la simultanéité provisionnée). Pour de plus amples informations, consultez Travailler avec la métrique `ClaimedAccountConcurrency`.

Métriques d'invocations asynchrones

Les métriques d'invocations asynchrones fournissent des informations détaillées sur les invocations asynchrones à partir des sources d'événements et des invocations directes. Vous pouvez définir des seuils et des alarmes pour vous avertir de certaines modifications. Par exemple, en cas d'augmentation indésirable du nombre d'événements mis en file d'attente pour traitement (`AsyncEventsReceived`). Ou encore lorsqu'un événement attend depuis longtemps d'être traité (`AsyncEventAge`).

- `AsyncEventsReceived` : le nombre d'événements que Lambda met correctement en file d'attente pour traitement. Cette métrique fournit des informations sur le nombre d'événements qu'une fonction Lambda reçoit. Surveillez cette métrique et définissez des alarmes pour des seuils afin de détecter les problèmes. Par exemple, pour détecter un nombre indésirable d'événements envoyés à Lambda et pour diagnostiquer rapidement les problèmes résultant de configurations de déclencheurs ou de fonctions incorrectes. Les incompatibilités entre `AsyncEventsReceived` et `Invocations` peuvent indiquer une disparité de traitement, des événements abandonnés ou un éventuel backlog dans les files d'attente.
- `AsyncEventAge` : le temps entre le moment où Lambda met correctement l'événement en file d'attente et le moment où la fonction est invoquée. La valeur de cette métrique augmente

lorsque des événements font l'objet d'une nouvelle tentative en raison d'échecs d'invocation ou de limitation. Surveillez cette métrique et définissez des alarmes pour les seuils sur différentes statistiques lorsqu'une accumulation de file d'attente se produit. Pour résoudre le problème d'une augmentation de cette métrique, examinez la métrique `Errors` pour identifier les erreurs de fonction et la métrique `Throttles` pour identifier les problèmes de simultanéité.

- `AsyncEventsDropped` : le nombre d'événements supprimés sans que la fonction ne soit exécutée correctement. Si vous configurez une file d'attente de lettres mortes (DLQ) ou une destination `OnFailure`, les événements y sont envoyés avant d'être supprimés. Les événements sont supprimés pour diverses raisons. Par exemple, les événements peuvent dépasser l'âge maximal de l'événement ou atteindre le nombre maximal de nouvelles tentatives, ou la simultanéité réservée peut être définie sur 0. Pour déterminer pourquoi les événements sont supprimés, examinez la métrique `Errors` pour identifier les erreurs de fonction et la métrique `Throttles` pour identifier les problèmes de simultanéité.

Métriques de mappage des sources d’événements

Les métriques de mappage des sources d’événements fournissent des informations sur le comportement de traitement de votre mappage des sources d’événements. Ces mesures vous aident à surveiller le flux et le statut des événements, y compris les événements que votre mappage des sources d’événements a correctement traités, filtrés ou supprimés.

Vous devez vous inscrire pour recevoir des statistiques relatives aux dénombrements (`PolledEventCount`, `FilteredOutEventCount`, `InvokedEventCount`, `FailedInvokeEventCount`, `DroppedEventCount`, `OnFailureDestinationDeliveredEventCount` et `DeletedEventCount`). Pour s’y inscrire, vous pouvez utiliser la console ou l’API Lambda.

Pour activer des métriques ou un mappage des sources d’événements (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez la fonction pour laquelle vous souhaitez activer les métriques.
3. Choisissez Configuration, puis Déclencheurs.
4. Choisissez le mappage des sources d’événements pour lequel vous souhaitez activer les métriques, puis choisissez Modifier.
5. Sous Configuration du mappage des sources d’événements, choisissez Activer les métriques.
6. Choisissez Enregistrer.

Vous pouvez également activer les métriques pour votre mappage des sources d'événements par programmation à l'aide de l'objet EventSourceMappingMetricsConfig dans votre EventSourceMappingConfiguration. Par exemple, la commande UpdateEventSourceMapping de la CLI ci-dessous active les métriques pour un mappage des sources d'événements :

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --metrics-config Metrics=EventCount
```

Affichez les statistiques relatives au nombre d'événements à l'aide de la statistique `Sum`.

> **Warning**
>
> Les mappages des sources d'événements Lambda traitent chaque événement au moins une fois, et le traitement des enregistrements peut être dupliqué. De ce fait, les événements peuvent être comptés plusieurs fois dans les métriques qui impliquent le décompte des événements.

- `PolledEventCount` : le nombre d'événements que Lambda lit correctement depuis la source des événements. Si Lambda interroge des événements, mais reçoit un sondage vide (aucun nouvel enregistrement), Lambda émet une valeur nulle pour cette métrique. Utilisez cette métrique pour détecter si votre mappage des sources d'événements interroge correctement les nouveaux événements.
- `FilteredOutEventCount` :pour le mappage des sources d'événements avec un critère de filtre, le nombre d'événements filtrés selon ces critères de filtre. Utilisez cette métrique pour détecter si votre mappage des sources d'événements filtre correctement les événements. Pour les événements qui répondent aux critères de filtre, Lambda émet une métrique nulle.
- `InvokedEventCount` : le nombre d'événements qui ont invoqué votre fonction Lambda. Utilisez cette métrique pour vérifier que les événements invoquent correctement votre fonction. Si un événement entraîne une erreur de fonctionnement ou une limitation, `InvokedEventCount` peut prendre en compte plusieurs fois pour le même événement interrogé en raison des nouvelles tentatives automatiques.
- `FailedInvokeEventCount` : le nombre d'événements pour lesquels Lambda a essayé d'invoquer votre fonction, mais qui ont échoué. Les appels peuvent échouer pour des raisons telles que des problèmes de configuration réseau, des autorisations incorrectes ou la suppression d'une fonction, d'une version ou d'un alias Lambda. Si l'option Réponse par lots partielle est activée pour

votre mappage des sources d'événements, `FailedInvokeEventCount` inclut tout événement dont la réponse contient un champ `BatchItemFailures` non vide.

> **Note**
>
> L'horodatage de la métrique `FailedInvokeEventCount` représente la fin de l'invocation de la fonction. Ce comportement est différent des autres métriques d'erreur d'appel Lambda, qui sont horodatées au début de l'invocation de la fonction.

- `DroppedEventCount` : le nombre d'événements que Lambda a abandonnés en raison de l'expiration ou de l'épuisement des tentatives. Plus précisément, il s'agit du nombre d'enregistrements qui dépassent les valeurs configurées pour `MaximumRecordAgeInSeconds` ou `MaximumRetryAttempts`. Il est important de noter que cela n'inclut pas le nombre d'enregistrements qui expirent en raison du dépassement des paramètres de conservation de la source de votre événement. Les événements abandonnés excluent également les événements que vous envoyez vers une destination en cas d'échec. Utilisez cette métrique pour détecter un arriéré croissant d'événements.
- `OnFailureDestinationDeliveredEventCount` : pour les mappages des sources d'événements avec une destination en cas d'échec configurée, le nombre d'événements envoyés vers cette destination. Utilisez cette métrique pour surveiller les erreurs de fonctionnement liées aux appels provenant de cette source d'événements. Si la livraison à destination échoue, Lambda gère les métriques comme suit :
  - Lambda n'émet pas la métrique `OnFailureDestinationDeliveredEventCount`.
  - Pour la métrique `DestinationDeliveryFailures`, Lambda émet un 1.
  - Pour la métrique `DroppedEventCount`, Lambda émet un nombre égal au nombre d'événements ayant échoué à la livraison.
- `DeletedEventCount` : le nombre d'événements que Lambda a réussi à supprimer après traitement. Si Lambda échoue à la tentative de suppression d'un événement, il émet une métrique nulle. Utilisez cette métrique pour vous assurer que les événements traités avec succès sont supprimés de votre source d'événements.

Si le mappage des sources d'événements est désactivé, vous ne recevrez pas de statistiques de mappage des sources d'événements. Des métriques peuvent également être manquantes si CloudWatch ou Lambda fait l'objet d'une disponibilité dégradée.

Les mesures de mappage des sources d'événements ne sont pas toutes disponibles pour chaque source d'événements. Actuellement, les métriques de mappage des sources d'événements sont disponibles pour les sources d'événements des flux Amazon SQS, Kinesis et DynamoDB. La matrice de disponibilité suivante récapitule les mesures prises en charge pour chaque type de source d'événement.

Métrique de mappage des sources d'événements	Prise en charge d'Amazon SQS	Prise en charge des flux Kinesis et DynamoDB
`PolledEventCount`	Oui	Oui
`FilteredOutEventCount`	Oui	Oui
`InvokedEventCount`	Oui	Oui
`FailedInvokeEventCount`	Oui	Oui
`DroppedEventCount`	Non	Oui
`OnFailureDestinationDeliveredEventCount`	Non	Oui
`DeletedEventCount`	Oui	Non

En outre, si votre mappage des sources d'événements est en mode alloué, Lambda fournit la métrique suivante :

- `ProvisionedPollers` : pour les mappages des sources d'événements en mode alloué, le nombre de sondeurs d'événements qui s'exécutent activement. Affichez cette métrique à l'aide de la métrique `MAX`.

Utilisation des journaux de fonction Lambda

Pour vous aider à résoudre les défaillances, surveille AWS Lambda automatiquement les fonctions Lambda en votre nom. Vous pouvez consulter les journaux des fonctions Lambda à l'aide de la console Lambda, de la console, du CloudWatch AWS Command Line Interface (AWS CLI) et de l'API. CloudWatch Vous pouvez également configurer Lambda pour envoyer les journaux à Amazon S3 et Firehose.

Tant que le rôle d'exécution de votre fonction dispose des autorisations nécessaires, Lambda capture les journaux de toutes les demandes traitées par votre fonction et les envoie à Amazon CloudWatch Logs, qui est la destination par défaut. Vous pouvez également laisser la console Lambda configurer Amazon S3 ou Firehose comme destinations de journalisation.

- CloudWatch Logs est la destination de journalisation par défaut pour les fonctions Lambda. CloudWatch Logs fournit des fonctionnalités de visualisation et d'analyse des journaux en temps réel, ainsi qu'une assistance pour créer des métriques et des alarmes basées sur les données de vos journaux.
- Amazon S3 est économique pour le stockage à long terme, et des services comme Athena peuvent être utilisés pour analyser les journaux. La latence est généralement plus élevée.
- Firehose propose une diffusion en continu gérée des journaux vers différentes destinations. Si vous devez envoyer des logs à d'autres AWS services (par exemple, OpenSearch Service ou Redshift Data API) ou à des plateformes tierces (comme Datadog, New Relic ou Splunk), Firehose simplifie ce processus en fournissant des intégrations prédéfinies. Vous pouvez également diffuser vers des points de terminaison HTTP personnalisés sans configurer d’infrastructure supplémentaire.

Choix d’une destination de service à laquelle envoyer les journaux

Tenez compte des facteurs clés suivants lorsque vous choisissez un service comme destination pour les journaux de fonction :

- La gestion des coûts varie selon le service. Amazon S3 constitue généralement l'option la plus économique pour le stockage à long terme, tandis que CloudWatch Logs vous permet de consulter les journaux, de traiter les journaux et de configurer des alertes en temps réel. Les coûts de Firehose incluent à la fois le service de diffusion en continu et le coût associé à la destination vers laquelle vous diffusez.
- Les capacités d'analyse varient d'un service à l'autre. CloudWatch Logs excelle dans la surveillance en temps réel et s'intègre nativement à d'autres CloudWatch fonctionnalités, telles que

Logs Insights et Live Tail. Amazon S3 fonctionne bien avec des outils d'analyse comme Athena, et peut s'intégrer à divers services, même s'il peut nécessiter une configuration supplémentaire. Firehose simplifie le streaming direct vers des AWS services spécifiques (tels que OpenSearch Service et Redshift Data API) et des plateformes tierces prises en charge (telles que Datadog et Splunk) en fournissant des intégrations prédéfinies, ce qui peut réduire le travail de configuration.

- La configuration et la facilité d'utilisation varient selon le service. CloudWatch Les journaux sont la destination par défaut des journaux. Elle fonctionne immédiatement, sans configuration supplémentaire, et permet de visualiser et d'analyser les journaux directement via la CloudWatch console. Si vous avez besoin d'envoyer les journaux à Amazon S3, vous devez effectuer une configuration initiale dans la console Lambda et configurer les autorisations du compartiment. Si vous avez besoin de journaux envoyés directement à des services tels que OpenSearch Service ou à des plateformes d'analyse tierces, Firehose peut simplifier ce processus.

Configuration des destinations de journal

AWS Lambda prend en charge plusieurs destinations pour vos journaux de fonctions. Ce guide explique les destinations de journalisation disponibles et vous aide à choisir l'option adaptée à vos besoins. Quelle que soit la destination choisie, Lambda propose des options pour contrôler le format, le filtrage et la diffusion des journaux.

Lambda prend en charge les formats JSON et texte brut pour les journaux de votre fonction. Les journaux structurés JSON offrent une meilleure facilité de recherche et permettent une analyse automatisée, tandis que les journaux en texte brut offrent une simplicité et des coûts de stockage potentiellement réduits. Vous pouvez contrôler les journaux que Lambda envoie à la destination que vous avez choisie en configurant les niveaux de journal pour les journaux du système et des applications. Le filtrage vous aide à gérer les coûts de stockage et trouver plus facilement les entrées de journal pertinentes lors du débogage.

Pour obtenir des instructions de configuration détaillées pour chaque destination, consultez les sections suivantes :

- Envoi automatique des journaux de la fonction Lambda à CloudWatch Logs
- Envoi de journaux de fonction Lambda à Firehose
- Envoi des journaux de fonction Lambda à Amazon S3

Configuration de commandes de journalisation avancées pour votre fonction Lambda

Pour vous permettre de mieux contrôler la manière dont les journaux de votre fonction sont capturés, traités et consommés, Lambda propose les options de configuration de journalisation suivantes :

- Format de journal : choisissez entre le format texte brut et le format JSON structuré pour les journaux de votre fonction.
- Niveau du journal : pour les journaux structurés JSON, choisissez le niveau de détail des journaux auxquels Lambda envoie CloudWatch, par exemple`FATAL`,`ERROR`,, `WARN` `INFO`DEBUG, et. `TRACE`
- Groupe de journaux : choisissez le groupe de CloudWatch journaux auquel votre fonction envoie les journaux.

Pour en savoir plus sur la configuration des contrôles de journalisation avancés, consultez les sections suivantes :

- Configuration des formats de journal JSON et en texte brut
- Filtrage au niveau du journal
- Configuration des groupes de CloudWatch journaux

Configuration des formats de journal JSON et en texte brut

La capture des sorties de votre journal sous forme de paires clé-valeur JSON facilite la recherche et le filtrage lors du débogage de vos fonctions. Avec les journaux au format JSON, vous pouvez également ajouter des balises et des informations contextuelles à vos journaux. Cela peut vous aider à effectuer une analyse automatique de gros volumes de données de journal. À moins que votre flux de développement ne repose sur des outils existants qui utilisent les journaux Lambda en texte brut, nous vous recommandons de sélectionner JSON pour le format de journal.

> **Instances gérées Lambda**
>
> Les instances gérées Lambda ne prennent en charge que le format de journal JSON. Lorsque vous créez une fonction Managed Instances, Lambda configure automatiquement le format du journal en JSON et vous ne pouvez pas le modifier en texte brut. Pour plus d'informations sur les instances gérées, consultezInstances gérées Lambda.

Pour tous les environnements d'exécution gérés par Lambda, vous pouvez choisir si les journaux système de votre fonction sont envoyés à CloudWatch Logs en texte brut non structuré ou au format JSON. Les journaux système sont les journaux générés par Lambda et sont parfois appelés journaux d'événements de plate-forme.

Pour les environnements d'exécution pris en charge, lorsque vous utilisez l'une des méthodes de journalisation intégrées prises en charge, Lambda peut également générer les journaux d'application de votre fonction (les journaux générés par votre code de fonction) au format JSON structuré. Lorsque vous configurez le format de journal de votre fonction pour ces environnements d'exécution, la configuration que vous choisissez s'applique à la fois aux journaux du système et aux journaux des applications.

Pour les environnements d'exécution pris en charge, si votre fonction utilise une bibliothèque ou une méthode de journalisation prise en charge, vous n'avez pas besoin de modifier votre code existant pour que Lambda capture les journaux au format JSON structuré.

> **Note**
>
> L'utilisation du format de journal JSON ajoute des métadonnées supplémentaires et code les messages de journal sous forme d'objets JSON contenant une série de paires clé-valeur. De ce fait, la taille des messages du journal de votre fonction peut augmenter.

Temps d'exécution et méthodes de journalisation pris en charge

Lambda prend actuellement en charge la possibilité de générer des journaux d'application structurés en JSON pour les environnements d'exécution suivants.

Language	Versions prises en charge
Java	Tous les environnements d'exécution Java sauf Java 8 sur Amazon Linux 1
.NET	.NET 8
Node.js	Node.js 16 et versions ultérieures
Python	Python 3.8 et versions ultérieures

Language	Versions prises en charge
Rust	N/A

Pour que Lambda envoie les journaux d'application de votre fonction CloudWatch au format JSON structuré, votre fonction doit utiliser les outils de journalisation intégrés suivants pour générer les journaux :

- Java : L'`LambdaLogger`enregistreur ou Log4j2. Pour de plus amples informations, veuillez consulter the section called “Logging”.
- .NET : `ILambdaLogger` instance de l'objet de contexte. Pour de plus amples informations, veuillez consulter Journalisation et surveillance des fonctions Lambda C#.
- Node.js - Les méthodes de console `console.trace`console.`debug`,`console.log`,`console.info`,`console.error`, et`console.warn`. Pour de plus amples informations, veuillez consulter the section called “Logging”.
- Python : `logging` bibliothèque Python standard. Pour de plus amples informations, veuillez consulter the section called “Logging”.
- Rust : La `tracing` caisse. Pour de plus amples informations, veuillez consulter the section called “Logging”.

Pour les autres environnements d’exécution Lambda gérés, Lambda ne prend actuellement en charge de manière native que la capture des journaux système au format JSON structuré. Cependant, vous pouvez toujours capturer les journaux d'applications au format JSON structuré dans n'importe quel environnement d'exécution en utilisant des outils de journalisation tels que Powertools pour générer AWS Lambda des sorties de journal au format JSON.

Formats de journal par défaut

Actuellement, le format de journal par défaut pour tous les environnements d’exécution Lambda est le texte brut. Pour les instances gérées Lambda, le format du journal est toujours JSON et ne peut pas être modifié.

Si vous utilisez déjà des bibliothèques de journalisation telles que Powertools AWS Lambda pour générer vos journaux de fonctions au format structuré JSON, vous n'avez pas besoin de modifier votre code si vous sélectionnez le formatage des journaux JSON. Lambda n’encode pas deux fois les

journaux déjà codés en JSON. Les journaux d'application de votre fonction continueront donc d'être capturés comme avant.

Format JSON pour les journaux du système

Lorsque vous définissez le format de journal de votre fonction sur JSON, chaque élément du journal système (événement de plate-forme) est capturé sous la forme d'un objet JSON contenant des paires clé-valeur avec les clés suivantes :

- `"time"` - heure à laquelle le message de journal a été généré
- `"type"` - type d'événement enregistré
- `"record"` - contenu de la sortie du journal

Le format de la valeur `"record"` varie en fonction du type d'événement enregistré. Pour de plus amples informations, veuillez consulter the section called "Types d'objets `Event` de l'API de télémétrie". Pour plus d'informations sur les niveaux de journalisation attribués aux événements du journal système, consultez the section called "Mappage des événements au niveau du journal système".

À titre de comparaison, les deux exemples suivants montrent le même résultat de journal à la fois au format texte brut et au format JSON structuré. Notez que dans la plupart des cas, les événements du journal système contiennent plus d'informations lorsqu'ils sont produits au format JSON que lorsqu'ils sont produits en texte brut.

Example texte brut :

```
2024-03-13 18:56:24.046000 fbe8c1   INIT_START  Runtime Version:
 python:3.12.v18  Runtime Version ARN: arn:aws:lambda:eu-
west-1::runtime:edb5a058bfa782cb9cedc6d534ac8b8c193bc28e9a9879d9f5ebaaf619cd0fc0
```

Example JSON structuré :

```
{
  "time": "2024-03-13T18:56:24.046Z",
  "type": "platform.initStart",
  "record": {
    "initializationType": "on-demand",
    "phase": "init",
    "runtimeVersion": "python:3.12.v18",
```

```
    "runtimeVersionArn": "arn:aws:lambda:eu-
west-1::runtime:edb5a058bfa782cb9cedc6d534ac8b8c193bc28e9a9879d9f5ebaaf619cd0fc0"
  }
}
```

> **Note**
>
> L'API the section called “API de télémétrie” toujours des événements de plate-forme tels que `START` et `REPORT` au format JSON. La configuration du format des journaux système auxquels Lambda envoie des messages CloudWatch n'affecte pas le comportement de l'API de télémétrie Lambda.

Format JSON pour les journaux d'applications

Lorsque vous configurez le format de journal de votre fonction au format JSON, les sorties du journal d'application écrites à l'aide des bibliothèques et méthodes de journalisation prises en charge sont capturées sous forme d'objet JSON contenant des paires clé-valeur avec les clés suivantes.

- `"timestamp"` - heure à laquelle le message de journal a été généré
- `"level"` - niveau de journalisation attribué au message
- `"message"` - contenu du message de journal
- `"requestId"` (Python, .NET et Node.js) ou `"AWSrequestId"` (Java) : l'ID de requête unique pour l'invocation de la fonction

Selon la méthode d'exécution et journalisation utilisée par votre fonction, cet objet JSON peut également contenir des paires de clés supplémentaires. Par exemple, dans Node.js, si votre fonction utilise des méthodes `console` pour enregistrer les objets d'erreur à l'aide de plusieurs arguments, l'objet JSON contiendra des paires clé-valeur supplémentaires avec les clés `errorMessage`, `errorType` et `stackTrace`. Pour en savoir plus sur les journaux au format JSON dans les différents environnements d'exécution Lambda, consultez the section called “Logging”, the section called “Logging” et the section called “Logging”.

> **Note**
>
> La clé utilisée par Lambda pour la valeur d'horodatage est différente pour les journaux système et les journaux des applications. Pour les journaux système, Lambda utilise

la clé `"time"` pour maintenir la cohérence avec l'API de télémétrie. Pour les journaux d'applications, Lambda suit les conventions des environnements d'exécution et utilise `"timestamp"`.

À titre de comparaison, les deux exemples suivants montrent le même résultat de journal à la fois au format texte brut et au format JSON structuré.

Example texte brut :

```
2024-10-27T19:17:45.586Z 79b4f56e-95b1-4643-9700-2807f4e68189 INFO some log message
```

Example JSON structuré :

```
{
    "timestamp":"2024-10-27T19:17:45.586Z",
    "level":"INFO",
    "message":"some log message",
    "requestId":"79b4f56e-95b1-4643-9700-2807f4e68189"
}
```

Configuration du format du journal de votre fonction

Pour configurer le format du journal pour votre fonction, vous pouvez utiliser la console Lambda ou le AWS Command Line Interface ()AWS CLI. Vous pouvez également configurer le format de journal d'une fonction à l'aide des commandes CreateFunctionet de l'API UpdateFunctionConfigurationLambda, de la AWS::Serverless::Functionressource AWS Serverless Application Model (AWS SAM) et de la CloudFormation AWS::Lambda::Functionressource.

La modification du format de journal de votre fonction n'affecte pas les journaux existants stockés dans CloudWatch Logs. Seuls les nouveaux journaux utiliseront le format mis à jour.

Si vous modifiez le format de journal de votre fonction en JSON sans définir le niveau de journal, Lambda définit automatiquement le niveau de journal d'application et le niveau de journal système de votre fonction sur INFO. Cela signifie que Lambda envoie uniquement des sorties de journal de niveau INFO ou inférieur à CloudWatch Logs. Pour en savoir plus sur le filtrage au niveau des journaux d'applications et de systèmes, consultez the section called “Filtrage au niveau du journal”

> **Note**
>
> Pour les environnements d'exécution Python, lorsque le format de journal de votre fonction est défini sur du texte brut, le paramètre de niveau de journal par défaut est WARN. Cela signifie que Lambda envoie uniquement des sorties de journal de niveau WARN ou inférieur à CloudWatch Logs. La modification du format de journal de votre fonction en JSON modifie ce comportement par défaut. Pour en savoir plus sur la journalisation dans Python, consultez the section called “Logging”.

Pour les fonctions Node.js qui émettent des journaux au format EMF (Embedded Metric Format), le fait de changer le format de journal de votre fonction en JSON peut CloudWatch empêcher la reconnaissance de vos métriques.

> **Important**
>
> Si votre fonction utilise Powertools for AWS Lambda (TypeScript) ou les bibliothèques clientes EMF open source pour émettre des journaux EMF, mettez à jour vos bibliothèques Powertools et EMF avec les dernières versions pour vous assurer qu'elles CloudWatch peuvent continuer à analyser correctement vos journaux. Si vous passez au format de journal JSON, nous vous recommandons également d’effectuer des tests pour garantir la compatibilité avec les métriques intégrées de votre fonction. Pour plus d’informations sur les fonctions node.js qui émettent des journaux EMF, consultez the section called “Utilisation de bibliothèques clientes au format métrique intégré (EMF) avec des journaux JSON structurés”.

Configurer le format de journal d’une fonction (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisir une fonction
3. Dans la page de configuration de la fonction, choisissez Outils de surveillance et d’exploitation.
4. Dans le volet de configuration de la journalisation, choisissez Modifier.
5. Sous Contenu du journal, pour Format du journal, sélectionnez Texte ou JSON.
6. Choisissez Enregistrer.

Pour modifier le format du journal d'une fonction existante (AWS CLI)

- Pour modifier le format de journalisation d'une fonction existante, utilisez la commande update-function-configuration. Définissez l'option `LogFormat` dans `LoggingConfig` sur `JSON` ou `Text`.

```
aws lambda update-function-configuration \
  --function-name myFunction \
  --logging-config LogFormat=JSON
```

Pour définir le format du journal lorsque vous créez une fonction (AWS CLI)

- Pour configurer le format du journal lorsque vous créez une fonction, utilisez l'option `--logging-config` de la commande create-function. Définissez `LogFormat` sur `JSON` ou `Text`. L'exemple de commande suivant crée une fonction Node.js qui génère des journaux au format JSON structuré.

  Si vous ne spécifiez pas de format de journal lorsque vous créez une fonction, Lambda utilisera le format de journal par défaut pour la version d'exécution que vous sélectionnez. Pour plus d'informations sur les formats de journalisation par défaut, consultez the section called "Formats de journal par défaut".

```
aws lambda create-function \
  --function-name myFunction \
  --runtime nodejs24.x \
  --handler index.handler \
  --zip-file fileb://function.zip \
  --role arn:aws:iam::123456789012:role/LambdaRole \
  --logging-config LogFormat=JSON
```

Filtrage au niveau du journal

Lambda peut filtrer les journaux de votre fonction afin que seuls les journaux d'un certain niveau de détail ou inférieur soient envoyés à CloudWatch Logs. Vous pouvez configurer le filtrage au niveau des journaux séparément pour les journaux système de votre fonction (les journaux générés par Lambda) et les journaux des applications (les journaux générés par le code de votre fonction).

Pour the section called “Temps d’exécution et méthodes de journalisation pris en charge”, vous n’avez pas besoin d’apporter de modifications au code de votre fonction pour que Lambda filtre les journaux d’application de votre fonction.

Pour tous les autres environnements d’exécution et méthodes de journalisation, le code de votre fonction doit générer les événements de journal vers `stdout` ou `stderr` sous forme d’objets au format JSON contenant une paire clé-valeur avec la clé `"level"`. Par exemple, Lambda interprète la sortie suivante vers `stdout` comme un journal de niveau DEBUG.

```
print('{"level": "debug", "msg": "my debug log", "timestamp":
 "2024-11-02T16:51:31.587199Z"}')
```

Si le champ de valeur `"level"` n’est pas valide ou est manquant, Lambda attribuera à la sortie du journal le niveau INFO. Pour que Lambda utilise le champ d'horodatage, vous devez spécifier le temps dans un format d'horodatage RFC 3339 valide. Si vous ne fournissez pas d’horodatage valide, Lambda attribuera au journal le niveau INFO et ajoutera un horodatage pour vous.

Lorsque vous nommez la clé d’horodatage, suivez les conventions du moteur d’exécution que vous utilisez. Lambda prend en charge les conventions de dénomination les plus courantes utilisées par les environnements d'exécution gérés.

> **Note**
>
> Pour utiliser le filtrage au niveau du journal, votre fonction doit être configurée pour utiliser le format de journal JSON. Le format de journal par défaut pour tous les environnements d’exécution Lambda est actuellement le texte brut. Pour savoir comment configurer le format de journal de votre fonction sur JSON, consultez the section called “Configuration du format du journal de votre fonction”.

Pour les journaux d’applications (les journaux générés par votre code de fonction), vous pouvez choisir entre les niveaux de journalisation suivants.

Niveau de journalisation	Utilisation standard
TRACE (le plus détaillé)	Les informations les plus précises utilisées pour tracer le chemin d’exécution de votre code

Niveau de journalisation	Utilisation standard
DEBUG	Informations détaillées pour le débogage du système
INFO	Messages qui enregistrent le fonctionnement normal de votre fonction
WARN	Messages relatifs à des erreurs potentielles susceptibles d'entraîner un comportement inattendu si elles ne sont pas traitées
ERROR	Messages concernant les problèmes qui empêchent le code de fonctionner comme prévu
FATAL (moindre détail)	Messages relatifs à des erreurs graves entraînant l'arrêt du fonctionnement de l'applica tion

Lorsque vous sélectionnez un niveau de journal, Lambda envoie les journaux à ce niveau et aux niveaux inférieurs à CloudWatch Logs. Par exemple, si vous définissez le niveau de journal d'application d'une fonction sur WARN, Lambda n'envoie pas de sorties de journal aux niveaux INFO et DEBUG. Le niveau de journal d'application par défaut pour le filtrage des journaux est INFO.

Lorsque Lambda filtre les journaux d'application de votre fonction, les messages de journal sans niveau se voient attribuer le niveau de journal INFO.

Pour les journaux système (les journaux générés par le service Lambda), vous pouvez choisir entre les niveaux de journalisation suivants.

Niveau de journalisation	Usage
DEBUG (le plus détaillé)	Informations détaillées pour le débogage du système
INFO	Messages qui enregistrent le fonctionnement normal de votre fonction

Niveau de journalisation	Usage
WARN (moindre détail)	Messages relatifs à des erreurs potentielles susceptibles d'entraîner un comportement inattendu si elles ne sont pas traitées

Lorsque vous sélectionnez un niveau de journal, Lambda envoie des journaux à ce niveau ou à un niveau inférieur. Par exemple, si vous définissez le niveau de journal système d'une fonction sur INFO, Lambda n'envoie pas de sorties de journal au niveau DEBUG.

Par défaut, Lambda définit le niveau de journalisation du système sur INFO. Avec ce paramètre, Lambda envoie `"start"` et `"report"` enregistre automatiquement des messages à. CloudWatch Pour recevoir des journaux système plus ou moins détaillés, définissez le niveau de journal sur DEBUG ou WARN. Pour consulter la liste des niveaux de journalisation auxquels Lambda mappe les différents événements du journal système, consultez. the section called "Mappage des événements au niveau du journal système"

Configuration du filtrage au niveau du journal

Pour configurer le filtrage au niveau du journal des applications et du système pour votre fonction, vous pouvez utiliser la console Lambda ou le (). AWS Command Line Interface AWS CLI Vous pouvez également configurer le niveau de journalisation d'une fonction à l'aide des commandes CreateFunctionet de l'API UpdateFunctionConfigurationLambda, de la AWS::Serverless::Functionressource AWS Serverless Application Model (AWS SAM) et de la CloudFormation AWS::Lambda::Functionressource.

Notez que si vous définissez le niveau de journalisation de votre fonction dans votre code, ce paramètre a priorité sur tous les autres paramètres de journalisation que vous configurez. Par exemple, si vous utilisez la méthode Python `logging` `setLevel()` pour définir le niveau de journalisation de votre fonction sur INFO, ce paramètre a priorité sur le paramètre WARN que vous configurez à l'aide de la console Lambda.

Pour configurer le niveau de journal de l'application ou du système d'une fonction existante (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Dans la page de configuration de la fonction, choisissez Outils de surveillance et d'exploitation.

4. Dans le volet de configuration de la journalisation, choisissez Modifier.
5. Sous Contenu du journal, pour le format du journal, assurez-vous que JSON est sélectionné.
6. À l'aide des boutons d'option, sélectionnez le niveau de journal des applications et le niveau de journal du système souhaités pour votre fonction.
7. Choisissez Enregistrer.

Pour configurer le niveau de journal de l'application ou du système d'une fonction existante (AWS CLI)

- Pour modifier le niveau du journal de l'application ou du système d'une fonction existante, utilisez la commande update-function-configuration. Utilisez `--logging-config` pour définir `SystemLogLevel` sur l'une des valeurs suivantes : `DEBUG`, `INFO` ou `WARN`. Défini `ApplicationLogLevel` sur l'une des valeurs `DEBUG`, `INFO`, `WARN`, `ERROR` ou `FATAL`.

```
aws lambda update-function-configuration \
  --function-name myFunction \
  --logging-config LogFormat=JSON,ApplicationLogLevel=ERROR,SystemLogLevel=WARN
```

Pour configurer le filtrage au niveau du journal lorsque vous créez une fonction

- Pour configurer le filtrage au niveau du journal lorsque vous créez une fonction, utilisez l'option `--logging-config` pour définir les clés `SystemLogLevel` et `ApplicationLogLevel` de la commande create-function. A défini `SystemLogLevel` sur l'une des valeurs `DEBUG`, `INFO` ou `WARN`. Défini `ApplicationLogLevel` sur l'une des valeurs `DEBUG`, `INFO`, `WARN`, `ERROR` ou `FATAL`.

```
aws lambda create-function \
  --function-name myFunction \
  --runtime nodejs24.x \
  --handler index.handler \
  --zip-file fileb://function.zip \
  --role arn:aws:iam::123456789012:role/LambdaRole \
  --logging-config LogFormat=JSON,ApplicationLogLevel=ERROR,SystemLogLevel=WARN
```

Mappage des événements au niveau du journal système

Pour les événements de journal au niveau du système générés par Lambda, le tableau suivant définit le niveau de journal attribué à chaque événement. Pour plus d'informations sur les événements répertoriés dans le tableau, consultez the section called "Référence du schéma `Event`"

Nom de l'événement	Condition	Niveau de journalisation attribué
initStart	runtimeVersion est définie	INFO
initStart	runtimeVersion est définie	DEBUG
initRuntimeDone	status=success	DEBUG
initRuntimeDone	status!=success	WARN
initReport	initializationType!=on-demand	INFO
initReport	initializationType=on-demand	DEBUG
initReport	status!=success	WARN
restoreStart	runtimeVersion est définie	INFO
restoreStart	runtimeVersion n'est pas définie	DEBUG
restoreRuntimeDone	status=success	DEBUG
restoreRuntimeDone	status!=success	WARN
restoreReport	status=success	INFO
restoreReport	status!=success	WARN
démarrer	-	INFO
runtimeDone	status=success	DEBUG
runtimeDone	status!=success	WARN

Nom de l'événement	Condition	Niveau de journalisation attribué
report	status=success	INFO
report	status!=success	WARN
Extension	status=success	INFO
Extension	status=success	WARN
LogSubscription	-	INFO
telemetrySubscription	-	INFO
logsDropped	-	WARN

> **Note**
>
> L'API the section called "API de télémétrie" toujours l'ensemble complet des événements de la plateforme. La configuration du niveau des journaux système auxquels Lambda envoie des messages CloudWatch n'affecte pas le comportement de l'API de télémétrie Lambda.

Filtrage au niveau du journal des applications avec des environnements d'exécution personnalisés

Lorsque vous configurez le filtrage au niveau du journal des applications pour votre fonction, Lambda définit en arrière-plan le niveau du journal des applications au moment de l'exécution à l'aide de la variable d'environnement `AWS_LAMBDA_LOG_LEVEL`. Lambda définit également le format du journal de votre fonction à l'aide de la variable d'environnement `AWS_LAMBDA_LOG_FORMAT`. Vous pouvez utiliser ces variables pour intégrer les commandes de journalisation avancées de Lambda dans un environnement d'exécution personnalisé.

Pour pouvoir configurer les paramètres de journalisation d'une fonction à l'aide d'un environnement d'exécution personnalisé avec la console Lambda et Lambda AWS CLI APIs, configurez votre environnement d'exécution personnalisé pour vérifier la valeur de ces variables d'environnement.

Vous pouvez ensuite configurer les enregistreurs de votre environnement d'exécution conformément au format de journal et aux niveaux de journal que vous avez sélectionnés.

Envoi automatique des journaux de la fonction Lambda à CloudWatch Logs

Par défaut, Lambda capture automatiquement les journaux de toutes les invocations de fonction et les envoie à CloudWatch Logs, à condition que le rôle d'exécution de votre fonction dispose des autorisations nécessaires. Par défaut, ces journaux sont stockés dans un groupe de journaux nommé /aws/lambda/*<nom-fonction>*. Pour améliorer le débogage, vous pouvez insérer des instructions de journalisation personnalisées dans votre code, que Lambda intégrera de manière transparente à CloudWatch Logs. Si nécessaire, vous pouvez configurer votre fonction pour envoyer les journaux à un groupe différent à l'aide de la console Lambda, AWS CLI, ou de l'API Lambda. Pour en savoir plus, veuillez consulter the section called "Configuration des journaux CloudWatch de fonctions".

Pour afficher les journaux des fonctions Lambda à l'aide de la console Lambda, utilisez la console CloudWatch, l'AWS Command Line Interface (AWS CLI) ou l'API CloudWatch. Pour plus d'informations, consultez Affichage CloudWatch des journaux pour les fonctions Lambda.

> **Note**
>
> L'affichage des journaux après l'invocation d'une fonction peut prendre de 5 à 10 minutes .

Autorisations IAM requises

Votre rôle d'exécution doit disposer d'une autorisation pour charger des journaux sur CloudWatch Logs :

- `logs:CreateLogGroup`
- `logs:CreateLogStream`
- `logs:PutLogEvents`

Pour de plus amples informations, consultez Using identity-based policies (IAM policies) for CloudWatch Logs dans le Guide de l'utilisateur Amazon CloudWatch.

Vous pouvez ajouter des autorisations CloudWatch Logs à l'aide de la politique `AWSLambdaBasicExecutionRole` gérée par AWS, fournie par Lambda. Exécutez la commande suivante pour ajouter cette politique à votre rôle :

```
aws iam attach-role-policy --role-name your-role --policy-arn arn:aws:iam::aws:policy/
service-role/AWSLambdaBasicExecutionRole
```

Pour de plus amples informations, consultez the section called "AWS politiques gérées".

Tarification

L'utilisation de journaux Lambda n'occasionne aucun coût supplémentaire. En revanche, les frais CloudWatch Logs standard s'appliquent. Pour plus d'informations, consultez Tarification CloudWatch.

Configuration des groupes de CloudWatch journaux

Par défaut, crée CloudWatch automatiquement un groupe de journaux `/aws/lambda/<function name>` portant le nom de votre fonction lors de son appel initial. Pour configurer votre fonction afin d'envoyer des journaux à un groupe de journaux existant, ou pour créer un nouveau groupe de journaux pour votre fonction, vous pouvez utiliser la console Lambda ou AWS CLI. Vous pouvez également configurer des groupes de journaux personnalisés à l'aide des commandes CreateFunctionet de l'API UpdateFunctionConfigurationLambda et de la ressource AWS Serverless Application Model (AWS SAM) AWS: :Serverless : :Function.

Vous pouvez configurer plusieurs fonctions Lambda pour envoyer des journaux au même groupe de CloudWatch journaux. Par exemple, vous pouvez utiliser un seul groupe de journaux pour stocker les journaux de toutes les fonctions Lambda qui constituent une application particulière. Lorsque vous utilisez un groupe de journaux personnalisé pour une fonction Lambda, les flux de journaux créés par Lambda incluent le nom et la version de la fonction. Cela garantit que le mappage entre les messages du journal et les fonctions est préservé, même si vous utilisez le même groupe de journaux pour plusieurs fonctions.

Le format de dénomination des flux de journaux pour les groupes de journaux personnalisés suit cette convention :

```
YYYY/MM/DD/<function_name>[<function_version>][<execution_environment_GUID>]
```

Notez que lors de la configuration d'un groupe de journaux personnalisé, le nom que vous sélectionnez pour votre groupe de journaux doit respecter les règles de dénomination CloudWatch des journaux. En outre, les noms de groupes de journaux personnalisés ne doivent pas commencer

par la chaîne `aws/`. Si vous créez un groupe de journaux personnalisé en commençant par `aws/`, Lambda ne sera pas en mesure de créer le groupe de journaux. Par conséquent, les journaux de votre fonction ne seront pas envoyés à CloudWatch.

Pour modifier le groupe de journaux d'une fonction (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Dans la page de configuration de la fonction, choisissez Outils de surveillance et d'exploitation.
4. Dans le volet de configuration de la journalisation, choisissez Modifier.
5. Dans le volet Groupe de journalisation, pour le groupe de CloudWatch journaux, sélectionnez Personnalisé.
6. Sous Groupe de journaux personnalisé, entrez le nom du groupe de CloudWatch journaux auquel votre fonction doit envoyer des journaux. Si vous entrez le nom d'un groupe de journaux existant, votre fonction utilisera ce groupe. S'il n'existe aucun groupe de journaux portant le nom que vous entrez, Lambda créera un nouveau groupe de journaux portant ce nom pour votre fonction.

Pour modifier le groupe de journaux d'une fonction (AWS CLI)

- Pour modifier le groupe de journaux d'une fonction existante, utilisez la commande update-function-configuration.

```
aws lambda update-function-configuration \
  --function-name myFunction \
  --logging-config LogGroup=myLogGroup
```

Pour indiquer un groupe de journaux personnalisé lorsque vous créez une fonction (AWS CLI)

- Pour spécifier un groupe de journaux personnalisé lorsque vous créez une nouvelle fonction Lambda à l'aide de AWS CLI, utilisez l'`--logging-config`option. L'exemple de commande suivant crée une fonction Lambda Node.js qui envoie des journaux à un groupe de journaux nommé `myLogGroup`.

```
aws lambda create-function \
  --function-name myFunction \
  --runtime nodejs24.x \
```

```
--handler index.handler \
--zip-file fileb://function.zip \
--role arn:aws:iam::123456789012:role/LambdaRole \
--logging-config LogGroup=myLogGroup
```

Autorisations du rôle d'exécution

Pour que votre fonction puisse envoyer des CloudWatch journaux à Logs, elle doit disposer de l'PutLogEventsautorisation logs :. Lorsque vous configurez le groupe de journaux de votre fonction à l'aide de la console Lambda, Lambda ajoute cette autorisation au rôle dans les conditions suivantes :

- La destination du service est définie sur CloudWatch Logs
- Le rôle d'exécution de votre fonction n'est pas autorisé à télécharger les journaux dans CloudWatch Logs (la destination par défaut)

> **Note**
>
> Lambda n'ajoute aucune autorisation Put pour les destinations de journal Amazon S3 ou Firehose.

Lorsque Lambda ajoute cette autorisation, elle autorise la fonction à envoyer des journaux à n'importe quel groupe de CloudWatch journaux Logs.

Pour empêcher Lambda de mettre automatiquement à jour le rôle d'exécution de la fonction et de le modifier manuellement, développez Autorisations et décochez Ajouter les autorisations requises.

Lorsque vous configurez le groupe de journaux de votre fonction à l'aide de AWS CLI, Lambda n'ajoute pas automatiquement l'`logs:PutLogEvents`autorisation. Ajoutez l'autorisation au rôle d'exécution de votre fonction si elle ne l'a pas déjà. Cette autorisation est incluse dans la politique AWSLambdaBasicExecutionRolegérée.

CloudWatch journalisation pour les instances gérées Lambda

Lorsque vous utilisez des instances gérées Lambda, d'autres considérations s'appliquent à l'envoi de journaux vers CloudWatch Logs :

Exigences de mise en réseau VPC

Les instances gérées Lambda s'exécutent sur des instances appartenant au client EC2 au sein de votre VPC. Pour envoyer des CloudWatch journaux à Logs et des traces à X-Ray, vous devez vous assurer qu' AWS APIs ils sont routables depuis votre VPC. Vous avez plusieurs options:

- AWS PrivateLink (recommandé) : AWS PrivateLinkà utiliser pour créer des points de terminaison VPC pour les services Logs CloudWatch et X-Ray. Cela permet à vos instances d'accéder à ces services en privé sans avoir besoin d'une passerelle Internet ou d'une passerelle NAT. Pour plus d'informations, consultez la section Utilisation CloudWatch des journaux avec les points de terminaison VPC de l'interface.
- Passerelle NAT : configurez une passerelle NAT pour autoriser l'accès Internet sortant à partir de vos sous-réseaux privés.
- Passerelle Internet : pour les sous-réseaux publics, assurez-vous qu'une passerelle Internet est configurée sur votre VPC.

Si CloudWatch Logs ou X-Ray ne APIs sont pas routables depuis votre VPC, vos journaux de fonctions et vos traces ne seront pas fournis.

Invocations simultanées et attribution de journaux

Les environnements d'exécution des instances gérées Lambda peuvent traiter plusieurs appels simultanément. Lorsque plusieurs appels sont exécutés simultanément, leurs entrées de journal sont entrelacées dans le même flux de journal. Pour filtrer et analyser efficacement les journaux provenant d'appels simultanés, vous devez vous assurer que chaque entrée de journal inclut l'ID de AWS demande.

Nous recommandons l'une des approches suivantes :

- Utiliser des enregistreurs d'exécution Lambda par défaut (recommandé) : Les bibliothèques de journalisation par défaut fournies par les environnements d'exécution gérés par Lambda incluent automatiquement l'ID de demande dans chaque entrée de journal.
- Implémenter une journalisation JSON structurée : si vous créez un environnement d'exécution personnalisé ou si vous avez besoin d'une journalisation personnalisée, implémentez des journaux au format JSON qui incluent l'ID de demande dans chaque entrée. Les instances gérées Lambda ne prennent en charge que le format de journal JSON. Incluez le `requestId` champ dans vos journaux JSON pour activer le filtrage par invocation :

```
{
  "timestamp": "2025-01-15T10:30:00.000Z",
  "level": "INFO",
  "requestId": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
  "message": "Processing request"
}
```

Grâce à l'attribution de l'ID de demande, vous pouvez filtrer CloudWatch les entrées du journal Logs pour un appel spécifique à l'aide des requêtes CloudWatch Logs Insights. Par exemple :

```
fields @timestamp, @message
| filter requestId = "a1b2c3d4-e5f6-7890-abcd-ef1234567890"
| sort @timestamp asc
```

Pour plus d'informations sur les exigences de journalisation des instances gérées Lambda, consultez. Comprendre l'environnement d'exécution des instances gérées Lambda

Affichage CloudWatch des journaux pour les fonctions Lambda

Vous pouvez consulter les CloudWatch journaux Amazon de votre fonction Lambda à l'aide de la console Lambda, de la CloudWatch console ou du (). AWS Command Line Interface AWS CLI Suivez les instructions dans les sections suivantes pour accéder aux journaux de votre fonction.

Diffusez les journaux des fonctions avec CloudWatch Logs Live Tail

Amazon CloudWatch Logs Live Tail vous aide à résoudre rapidement les problèmes liés à vos fonctions en affichant une liste de diffusion des nouveaux événements de journal directement dans la console Lambda. Vous pouvez afficher et filtrer les journaux ingérés depuis votre fonction Lambda en temps quasi réel, ce qui vous permet de détecter et de résoudre rapidement les problèmes.

> **Note**
>
> Le coût des sessions Live Tail est calculé en fonction du temps d'utilisation de la session, par minute. Pour plus d'informations sur les tarifs, consultez Amazon CloudWatch Pricing.

Comparaison entre Live Tail et --log-type Tail

Il existe plusieurs différences entre CloudWatch Logs Live Tail et l'option LogType: Tail de l'API Lambda (`--log-type Tail`dans le AWS CLI) :

- `--log-type Tail` ne renvoie que les quatre premiers Ko des journaux d'invocation. Live Tail ne partage pas cette limite et peut recevoir jusqu'à 500 événements de journal par seconde.
- `--log-type Tail` capture et envoie les journaux avec la réponse, ce qui peut avoir un impact sur la latence de réponse de la fonction. Live Tail n'affecte pas la latence de réponse des fonctions.
- `--log-type Tail` ne prend en charge que les invocations synchrones. Live Tail fonctionne à la fois pour les invocations synchrones et asynchrones.

> **Note**
>
> Les instances gérées Lambda ne prennent pas en charge cette option. `--log-type Tail` Utilisez CloudWatch Logs Live Tail ou interrogez directement CloudWatch les journaux pour consulter les journaux des fonctions Managed Instances.

Permissions

Les autorisations suivantes sont requises pour démarrer et arrêter les sessions CloudWatch Logs Live Tail :

- `logs:DescribeLogGroups`
- `logs:StartLiveTail`
- `logs:StopLiveTail`

Démarrage d'une session Live Tail dans la console Lambda

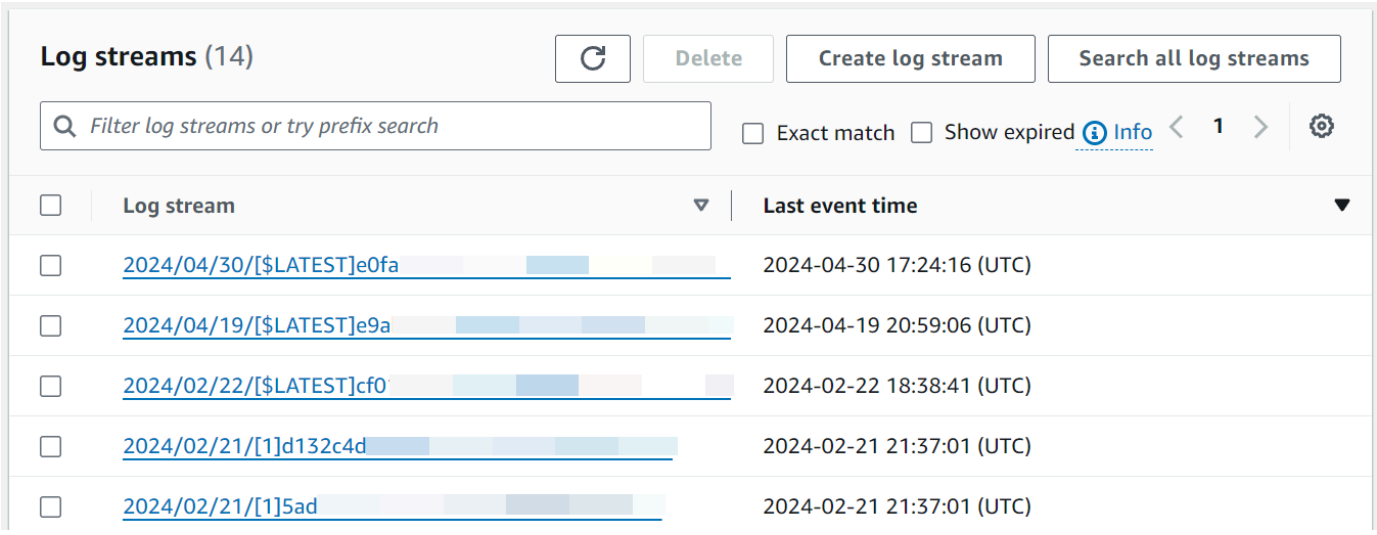
1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez le nom de la fonction.
3. Choisissez l'onglet Test.
4. Dans le volet Test event, sélectionnez CloudWatch Logs Live Tail.
5. Pour Sélectionner des groupes de journaux, le groupe de journaux de la fonction est sélectionné par défaut. Vous pouvez sélectionner jusqu'à cinq groupes de journaux à la fois.

6. (Facultatif) Pour m'afficher que les événements du journal contenant certains mots ou d'autres chaînes de caractères, saisissez le mot ou la chaîne dans la zone Ajouter un modèle de filtre. Le champ des filtres est sensible à la casse. Vous pouvez inclure plusieurs termes et opérateurs de motifs dans ce champ, y compris des expressions régulières (regex). Pour plus d'informations sur la syntaxe des modèles, consultez la section Syntaxe des modèles de filtres dans le guide de l'utilisateur Amazon CloudWatch Logs.
7. Sélectionnez Démarrer. Les événements du journal correspondants commencent à apparaître dans la fenêtre.
8. Pour arrêter la session Live Tail, choisissez Arrêter.

> **Note**
>
> La session Live Tail s'arrête automatiquement après 15 minutes d'inactivité ou lorsque la session de la console Lambda expire.

Accès aux journaux de fonctions à l'aide de la console

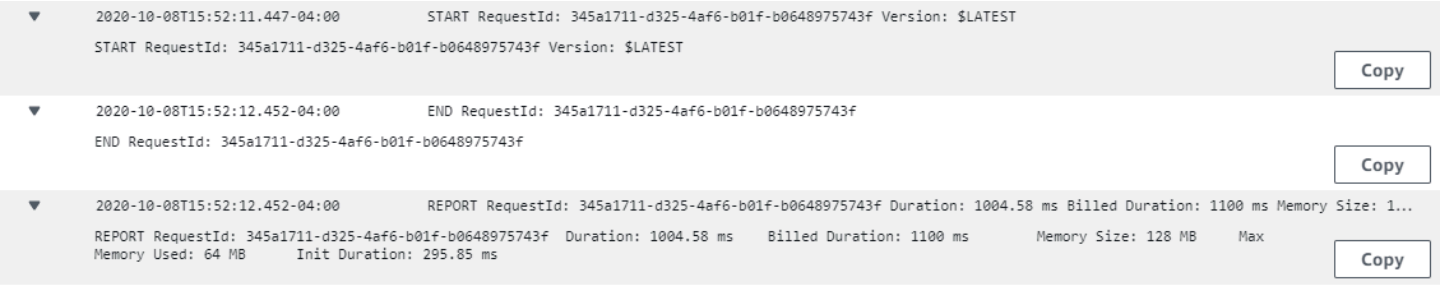
1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Sélectionnez une fonction.
3. Choisissez l'onglet Surveiller.
4. Choisissez Afficher CloudWatch les journaux pour ouvrir la CloudWatch console.
5. Faites défiler la page vers le bas et choisissez le flux de journaux pour les invocations de fonctions que vous souhaitez consulter.



Chaque instance d'une fonction Lambda possède un flux de journaux dédié. Si une fonction augmente verticalement, chaque instance simultanée possède son propre flux de journaux. Chaque fois qu'un nouvel environnement d'exécution est créé en réponse à une invocation, cela génère un nouveau flux de journaux. La convention de dénomination pour les flux de journaux est la suivante :

```
YYYY/MM/DD[Function version][Execution environment GUID]
```

Un environnement d'exécution unique écrit dans le même flux de journaux pendant sa durée de vie. Le flux de journaux contient les messages provenant de cet environnement d'exécution, ainsi que toute sortie du code de votre fonction Lambda. Chaque message est horodaté, y compris vos journaux personnalisés. Même si votre fonction ne journalise aucune sortie de votre code, trois instructions de journal minimales sont générées par invocation (START, END et REPORT) :



Ces journaux indiquent :

- RequestId— il s'agit d'un identifiant unique généré par demande. Si la fonction Lambda relance une requête, cet identifiant ne change pas et apparaît dans les journaux à chaque nouvelle tentative suivante.
- Début/Fin : ces éléments marquent une seule invocation, de sorte que chaque ligne de journal située entre eux appartient à la même invocation.
- Durée : durée totale d'invocation de la fonction de gestionnaire, à l'exception du code `INIT`.
- Durée facturée : applique la logique d'arrondi à des fins de facturation.
- Memory Size : la quantité de mémoire allouée à la fonction.
- Mémoire maximale utilisée : quantité de mémoire maximale utilisée lors de l'invocation.
- Durée d'initialisation : temps nécessaire pour exécuter la section `INIT` de code, en dehors du gestionnaire principal.

Accédez aux journaux avec AWS CLI

AWS CLI Il s'agit d'un outil open source qui vous permet d'interagir avec les AWS services à l'aide de commandes dans votre interface de ligne de commande. Pour effectuer les étapes de cette section, vous devez disposer de la version 2 de l'AWS CLI.

Vous pouvez utiliser AWS CLI pour récupérer les journaux d'une invocation à l'aide de l'option de commande `--log-type`. La réponse inclut un champ `LogResult` qui contient jusqu'à 4 Ko de journaux codés en base64 provenant de l'invocation.

Example récupérer un ID de journal

L'exemple suivant montre comment récupérer un ID de journal à partir du champ `LogResult` d'une fonction nommée `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Vous devriez voir la sortie suivante:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example décoder les journaux

Dans la même invite de commandes, utilisez l'utilitaire `base64` pour décoder les journaux. L'exemple suivant montre comment récupérer les journaux encodés en base64 pour `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

Vous devriez voir la sortie suivante :

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

L'utilitaire `base64` est disponible sous Linux, macOS et Ubuntu sous Windows. Les utilisateurs de macOS auront peut-être besoin d'utiliser `base64 -D`.

Example Script get-logs.sh

Dans la même invite de commandes, utilisez le script suivant pour télécharger les cinq derniers événements de journalisation. Le script utilise `sed` pour supprimer les guillemets du fichier de sortie et attend 15 secondes pour permettre la mise à disposition des journaux. La sortie comprend la réponse de Lambda, ainsi que la sortie de la commande `get-log-events`.

Copiez le contenu de l'exemple de code suivant et enregistrez-le dans votre répertoire de projet Lambda sous `get-logs.sh`.

L'cli-binary-formatoption est obligatoire si vous utilisez AWS CLI la version 2. Pour faire de ce paramètre le paramètre par défaut, exécutez `aws configure set cli-binary-format raw-in-base64-out`. Pour plus d'informations, consultez les options de ligne de commande globales AWS CLI prises en charge dans le Guide de l'utilisateur AWS Command Line Interface version 2.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS et Linux (uniquement)

Dans la même invite de commandes, les utilisateurs macOS et Linux peuvent avoir besoin d'exécuter la commande suivante pour s'assurer que le script est exécutable.

```
chmod -R 755 get-logs.sh
```

Example récupérer les cinq derniers événements de journal

Dans la même invite de commande, exécutez le script suivant pour obtenir les cinq derniers événements de journalisation.

```
./get-logs.sh
```

Vous devriez voir la sortie suivante :

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
```

```
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

Analyse des journaux et journalisation structurée

Avec CloudWatch Logs Insights, vous pouvez rechercher et analyser les données des journaux à l'aide d'une syntaxe de requête spécialisée. Le service exécute des requêtes sur plusieurs groupes de journaux et fournit un filtrage puissant grâce à la correspondance de modèles glob ou d'expressions régulières.

Vous pouvez tirer parti de ces fonctionnalités en implémentant une journalisation structurée dans vos fonctions Lambda. La journalisation structurée organise vos journaux dans un format prédéfini, ce qui permet de les interroger plus facilement. L'utilisation des niveaux de journaux est une première étape importante pour générer des journaux filtrables qui séparent les messages d'information des avertissements ou des erreurs. Prenons le code Node.js suivant :

```
exports.handler = async (event) => {
    console.log("console.log - Application is fine")
    console.info("console.info - This is the same as console.log")
    console.warn("console.warn - Application provides a warning")
    console.error("console.error - An error occurred")
}
```

Le fichier CloudWatch journal obtenu contient un champ distinct spécifiant le niveau de journalisation :

```
START RequestId: 99d91f9b-2dff-40ad-b9c8-664020094109 Version: $LATEST
2020-10-22T12:21:28.268Z        99d91f9b-2dff-40ad-b9c8-664020094109    INFO    console.log - Application is fine
2020-10-22T12:21:28.268Z        99d91f9b-2dff-40ad-b9c8-664020094109    INFO    console.info - This is the same as console.log
2020-10-22T12:21:28.268Z        99d91f9b-2dff-40ad-b9c8-664020094109    WARN    console.warn - Application provides a warning
2020-10-22T12:21:28.268Z        99d91f9b-2dff-40ad-b9c8-664020094109    ERROR   console.error - An error occurred
END RequestId: 99d91f9b-2dff-40ad-b9c8-664020094109
REPORT RequestId: 99d91f9b-2dff-40ad-b9c8-664020094109  Duration: 3.13 ms       Billed Duration: 100 ms Memory Size: 128 MB
Max Memory Used: 64 MB  Init Duration: 142.18 ms
```

Une requête CloudWatch Logs Insights peut ensuite filtrer au niveau du journal. Par exemple, pour rechercher uniquement les erreurs, vous pouvez utiliser la requête suivante :

```
fields @timestamp, @message
| filter @message like /ERROR/
| sort @timestamp desc
```

Journalisation structurée JSON

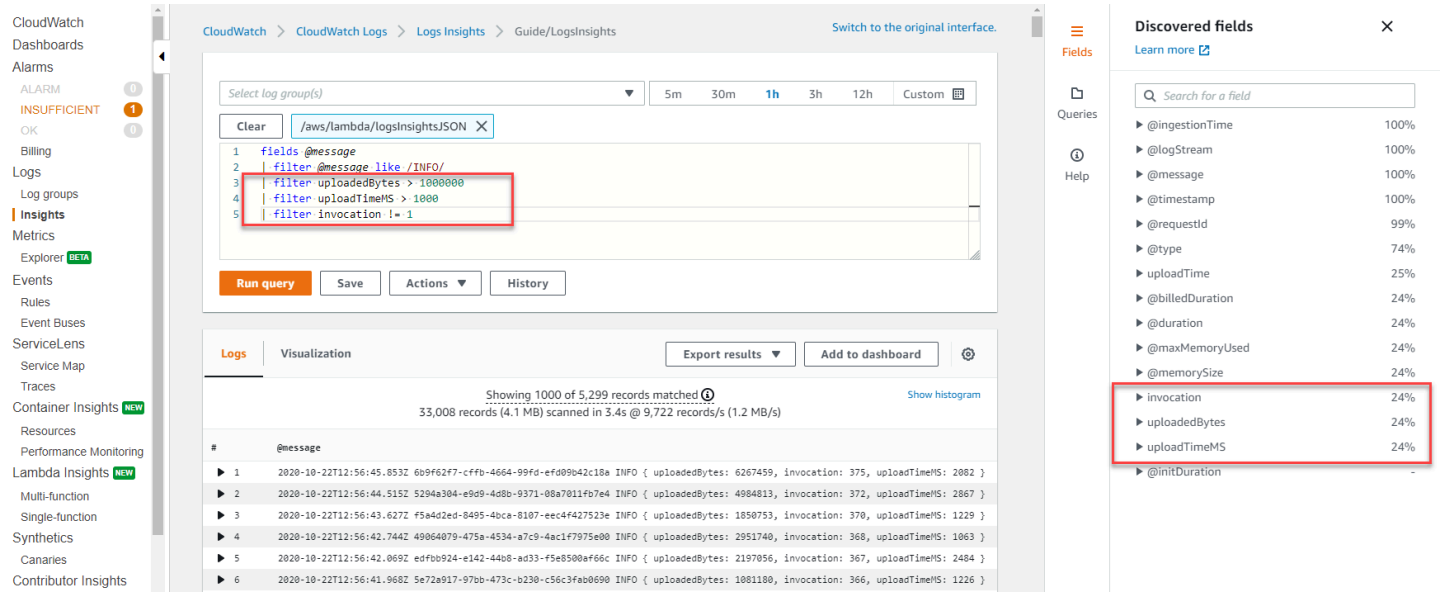
Le format JSON est couramment utilisé pour fournir une structure aux journaux d'applications. Dans l'exemple suivant, les journaux ont été convertis en JSON pour générer trois valeurs distinctes :

```
START RequestId: 22fda338-7b46-4c49-9531-efd6e8568480 Version: $LATEST
2020-10-22T11:35:59.216Z        22fda338-7b46-4c49-9531-efd6e8568480    INFO    { uploadedBytes: 5453396,
invocation: 5, uploadTimeMS: 4519 }
END RequestId: 22fda338-7b46-4c49-9531-efd6e8568480
REPORT RequestId: 22fda338-7b46-4c49-9531-efd6e8568480  Duration: 1.25 ms       Billed Duration: 100 ms Memory
Size: 128 MB    Max Memory Used: 65 MB
```

La fonctionnalité CloudWatch Logs Insights découvre automatiquement les valeurs dans la sortie JSON et analyse les messages sous forme de champs, sans avoir besoin d'une expression globale ou régulière personnalisée. À l'aide des journaux structurés JSON, la requête suivante trouve les invocations pour lesquelles le fichier chargé était supérieur à 1 Mo, le temps de chargement était supérieur à 1 seconde et l'invocation n'était pas un démarrage à froid :

```
fields @message
| filter @message like /INFO/
| filter uploadedBytes > 1000000
| filter uploadTimeMS > 1000
| filter invocation != 1
```

Cette requête peut produire le résultat suivant :



Les champs découverts en JSON sont automatiquement renseignés dans le menu Champs découvert à droite. Les champs standard émis par le service Lambda comportent le préfixe « @ » et vous pouvez effectuer des requêtes sur ces champs de la même manière. Les logs Lambda incluent toujours les champs @timestamp, @logStream, @message, @requestId, @duration, @billedDuration, @type, @maxMemoryUsed, @memorySize. Si X-Ray est activé pour une fonction, les journaux incluent également @ xrayTraceId et @xraySegmentId.

Lorsqu'une source d' AWS événement telle qu'Amazon S3, Amazon SQS ou Amazon EventBridge invoque votre fonction, l'événement complet est fourni à la fonction sous forme d'entrée d'objet JSON. En enregistrant cet événement dans la première ligne de la fonction, vous pouvez ensuite effectuer une requête sur l'un des champs imbriqués à l'aide de CloudWatch Logs Insights.

Requêtes Insights utiles

Le tableau suivant présente des exemples de requêtes Insights qui peuvent être utiles pour surveiller les fonctions Lambda.

Description	Exemple de syntaxe de requête

Les 100 dernières erreurs	`fields Timestamp, LogLevel, Message`
	`| filter LogLevel == "ERR"`
	`| sort @timestamp desc`
	`| limit 100`

Les 100 invocations les plus facturées	`filter @type = "REPORT"`

Description	Exemple de syntaxe de requête

	`| fields @requestId, @billedDuration`
	`| sort by @billedDuration desc`
	`| limit 100`

Pourcentage de démarrages à froid dans le nombre total d'invocations	`filter @type = "REPORT"`
	`| stats sum(strcontains(@message,`
	` "Init Duration"))/count(*) * 100 as`
	`  coldStartPct, avg(@duration)`
	`  by bin(5m)`

Rapport de percentile sur la durée Lambda	`filter @type = "REPORT"`
	`| stats`
	`  avg(@billedDuration) as Average,`
	`  percentile(@billedDuration, 99)`
	` as NinetyNinth,`
	`  percentile(@billedDuration, 95)`
	` as NinetyFifth,`
	`  percentile(@billedDuration, 90)`
	` as Ninetieth`
	`  by bin(30m)`

Rapport de percentile sur l'utilisation de la mémoire Lambda	`filter @type="REPORT"`
	`| stats avg(@maxMemoryUsed/1024/102`
	`4) as mean_MemoryUsed,`
	`  min(@maxMemoryUsed/1024/1024) as`
	` min_MemoryUsed,`
	`  max(@maxMemoryUsed/1024/1024) as`
	` max_MemoryUsed,`
	`  percentile(@maxMemoryUsed/1`
	`024/1024, 95) as Percentile95`

Invocations utilisant 100 % de la mémoire assignée	`filter @type = "REPORT" and @maxMemor`
	`yUsed=@memorySize`
	`| stats`
	`  count_distinct(@requestId)`
	`  by bin(30m)`

Description	Exemple de syntaxe de requête

Mémoire moyenne utilisée lors des invocations	`avgMemoryUsedPERC,`
	`    avg(@billedDuration) as avgDurati`
	`onMS`
	`    by bin(5m)`

Visualisation des statistiques de mémoire	`filter @type = "REPORT"`
	`| stats`
	`    max(@maxMemoryUsed / 1024 / 1024)`
	` as maxMemMB,`
	`    avg(@maxMemoryUsed / 1024 / 1024)`
	` as avgMemMB,`
	`    min(@maxMemoryUsed / 1024 / 1024)`
	` as minMemMB,`
	`    (avg(@maxMemoryUsed / 1024 /`
	` 1024) / max(@memorySize / 1024 /`
	` 1024)) * 100 as avgMemUsedPct,`
	`    avg(@billedDuration) as avgDurati`
	`onMS`
	`    by bin(30m)`

Invocations pour lesquelles Lambda s’est fermé	`filter @message like /Process exited/` `| stats count() by bin(30m)`
Invocations dont le délai a expiré	`filter @message like /Task timed out/` `| stats count() by bin(30m)`

Rapport de latence	`filter @type = "REPORT"`
	`| stats avg(@duration), max(@dura`
	`tion), min(@duration)`
	`  by bin(5m)`

Description	Exemple de syntaxe de requête
Mémoire surallouée	`filter @type = "REPORT" | stats max(@memorySize / 1024 / 1024) as provisonedMemMB, min(@maxMemoryUsed / 1024 / 1024) as smallestMemReqMB, avg(@maxMemoryUsed / 1024 / 1024) as avgMemUsedMB, max(@maxMemoryUsed / 1024 / 1024) as maxMemUsedMB, provisonedMemMB - maxMemUsedMB as overProvisionedMB`

Visualisation des journaux et tableaux de bord

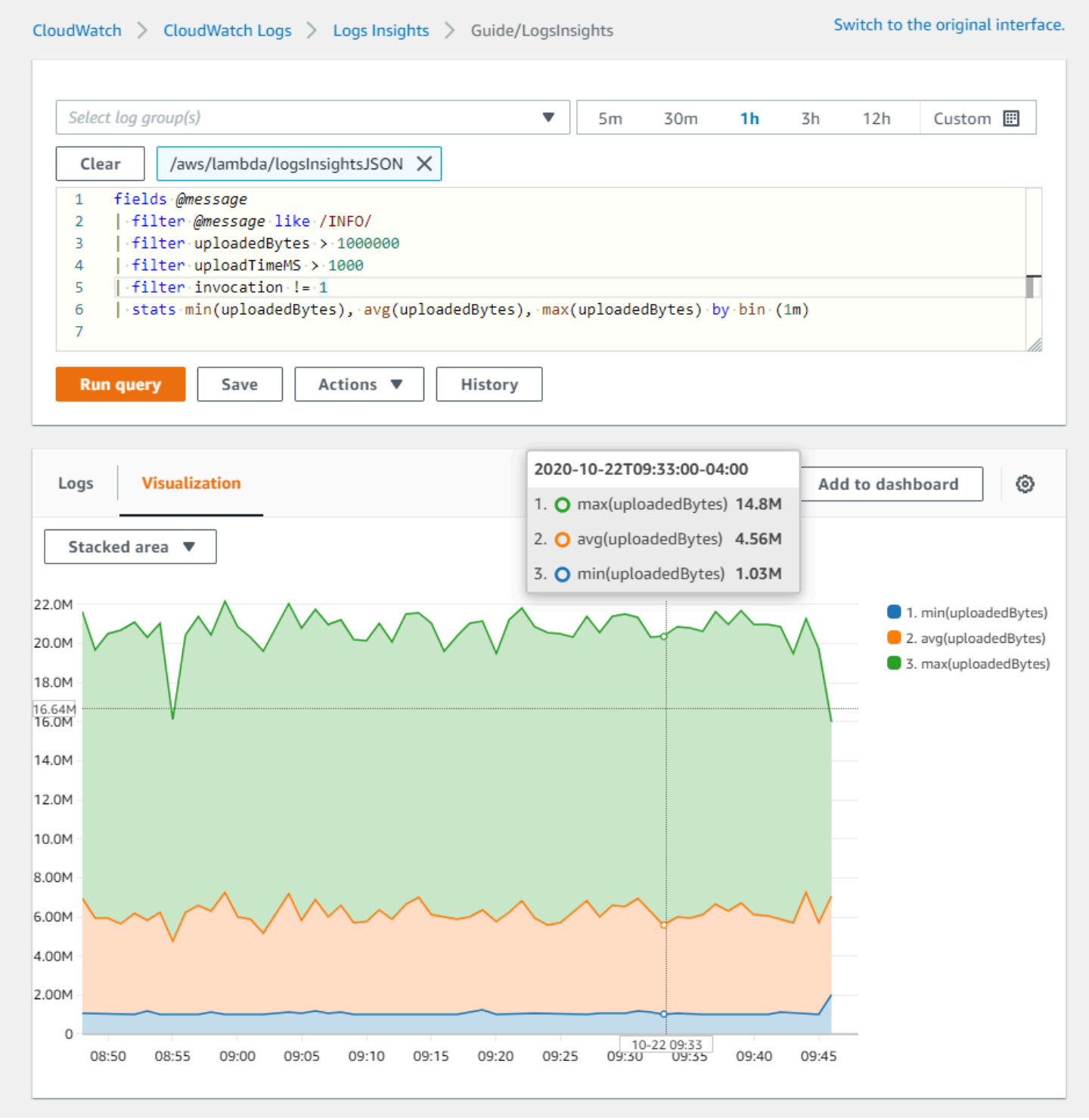
Pour toute requête CloudWatch Logs Insights, vous pouvez exporter les résultats au format Markdown ou CSV. Dans certains cas, il peut être plus utile de créer des visualisations à partir de requêtes, à condition qu'il existe au moins une fonction d'agrégation. La fonction `stats` vous permet de définir des agrégations et des regroupements.

L'exemple logInsightsJSON précédent filtrait en fonction de la taille et de la durée du chargement et excluait les premières invocations. Cela générait un tableau de données. Pour surveiller un système de production, il peut être plus utile de visualiser les tailles de fichier minimales, maximales et moyennes afin de détecter les valeurs aberrantes. Pour ce faire, appliquez la fonction stats avec les agrégats requis et effectuez un regroupement en fonction d'une valeur temporelle, par exemple chaque minute :

Par exemple, réfléchissez à la requête suivante. Il s'agit du même exemple de requête que celui de la section the section called "Journalisation structurée JSON", mais avec des fonctions d'agrégation supplémentaires :

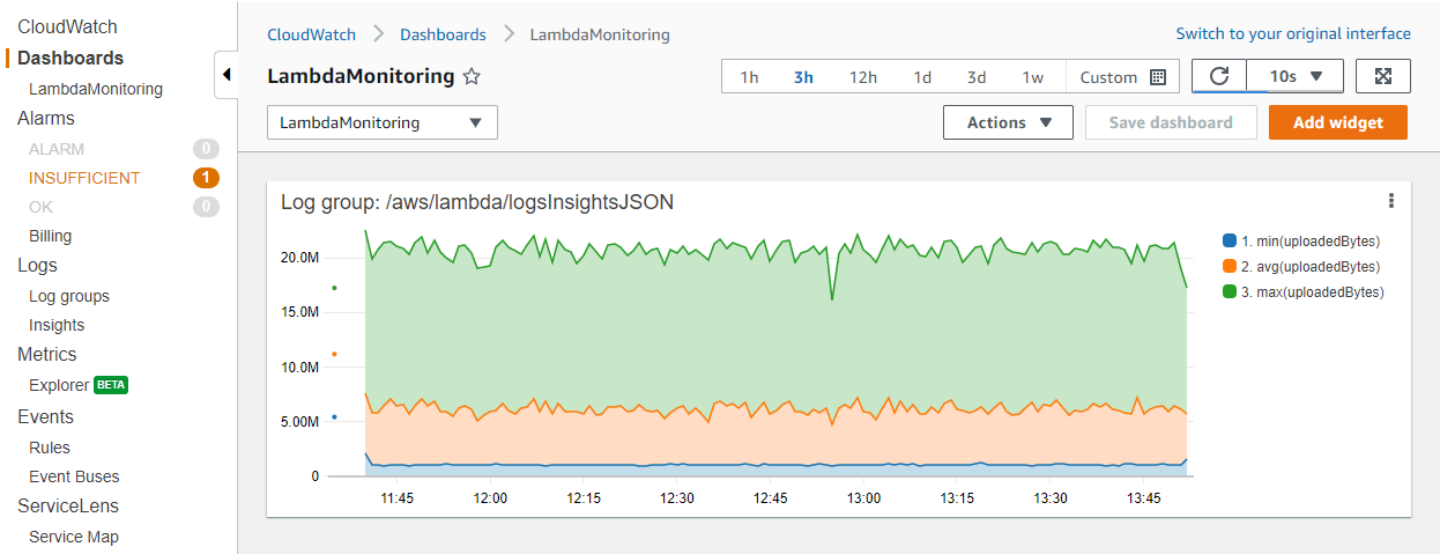
```
fields @message
| filter @message like /INFO/
| filter uploadedBytes > 1000000
| filter uploadTimeMS > 1000
| filter invocation != 1
| stats min(uploadedBytes), avg(uploadedBytes), max(uploadedBytes) by bin (1m)
```

Nous avons inclus ces agrégats, car il peut être plus utile de visualiser les tailles de fichier minimales, maximales et moyennes afin de détecter les valeurs aberrantes. Vous pouvez voir les résultats dans l'onglet Visualisation :



Une fois que vous avez terminé de créer la visualisation, vous pouvez éventuellement ajouter le graphique à un CloudWatch tableau de bord. Pour ce faire, choisissez Ajouter au tableau de bord

au-dessus de la visualisation. Cela ajoute la requête sous forme de widget et vous permet de sélectionner des intervalles d'actualisation automatiques, ce qui facilite le suivi continu des résultats :



Envoi de journaux de fonction Lambda à Firehose

La console Lambda offre désormais la possibilité d'envoyer des journaux de fonction à Firehose. Cela permet de diffuser en temps réel vos journaux vers diverses destinations prises en charge par Firehose, notamment des outils d'analytique tiers et des points de terminaison personnalisés.

> **Note**
>
> Vous pouvez configurer les journaux de fonction Lambda à envoyer à Firehose à l'aide de la console Lambda, AWS CLI, AWS CloudFormation et de tous les SDK AWS.

Tarification

Pour plus d'informations de tarification, consultez Tarification d'Amazon CloudWatch.

Autorisations requises pour la destination du journal Firehose

Lorsque vous utilisez la console Lambda pour configurer Firehose comme destination de journal de votre fonction, vous devez :

1. Avoir les autorisations IAM requises pour utiliser CloudWatch Logs avec Lambda.
2. Configurer des filtres d'abonnement avec Firehose. Ce filtre définit quels événements de journal sont fournis à votre flux Firehose.

Envoi de journaux de fonction Lambda à Firehose

Dans la console Lambda, vous pouvez envoyer des journaux de fonction directement à Firehose après avoir créé une nouvelle fonction. Pour ce faire, effectuez ces étapes :

1. Connectez-vous à Gestion de la console AWS et ouvrez la console Lambda.
2. Choisissez le nom de votre fonction.
3. Cliquez sur l'onglet Configuration.
4. Choisissez l'onglet Outils de surveillance et d'exploitation.
5. Dans le volet « Configuration de la journalisation », choisissez Modifier.
6. Dans la section « Contenu du journal », sélectionnez un format de journal.
7. Dans la section « Destination du journal » procédez comme suit :
   a. Sélectionnez un service de destination.
   b. Choisissez Créer un nouveau groupe de journaux ou utilisez un Groupe de journaux existant.

      > **Note**
      >
      > Si vous choisissez un groupe de journaux existant pour une destination Firehose, assurez-vous que le groupe de journaux que vous choisissez est un type de groupe de journaux `Delivery`.

   c. Choisissez un flux Firehose.
   d. Le groupe de journaux `Delivery` CloudWatch s'affichera.
8. Choisissez Enregistrer.

> **Note**
>
> Si le rôle IAM fourni dans la console ne dispose pas de l'autorisation requise, la configuration de la destination échouera. Pour résoudre ce problème, reportez-vous à Autorisations requises pour la destination du journal Firehose afin de fournir les autorisations requises.

Journalisation entre comptes

Vous pouvez configurer Lambda pour envoyer des journaux au flux de diffusion Firehose via un autre compte AWS. Cela nécessite de définir une destination et de configurer les autorisations appropriées dans les deux comptes.

Pour obtenir des instructions détaillées sur la configuration de la journalisation entre comptes, y compris les rôles et politiques IAM requis, consultez Configuration d'un nouvel abonnement entre comptes dans la documentation CloudWatch Logs.

Envoi des journaux de fonction Lambda à Amazon S3

Vous pouvez configurer votre fonction Lambda pour envoyer des journaux directement à Amazon S3 à l'aide de la console Lambda. Cette fonctionnalité fournit une solution rentable pour le stockage des journaux à long terme et permet d'utiliser de puissantes options d'analyse à l'aide de services comme Athena.

> **Note**
>
> Vous pouvez configurer les journaux de fonction Lambda à envoyer à Amazon S3 à l'aide de la console Lambda, AWS CLI, AWS CloudFormation et de tous les SDK AWS.

Tarification

Pour plus d'informations de tarification, consultez Tarification d'Amazon CloudWatch.

Autorisations requises pour la destination du journal Amazon S3

Lorsque vous utilisez la console Lambda pour configurer Amazon S3 comme destination de journal de votre fonction, vous devez :

1. Avoir les autorisations IAM requises pour utiliser CloudWatch Logs avec Lambda.
2. Pour Configurer un filtre d'abonnements CloudWatch Logs pour envoyer les journaux de fonction Lambda à Amazon S3. Ce filtre définit quels événements de journal sont fournis à votre compartiment Amazon S3.

Configurer un filtre d'abonnements CloudWatch Logs pour envoyer les journaux de fonction Lambda à Amazon S3

Pour envoyer des journaux depuis CloudWatch Logs vers Amazon S3, vous devez créer un filtre d'abonnement. Ce filtre définit quels événements de journal sont fournis à votre compartiment Amazon S3. Le groupe de journaux doit se trouver dans la même région que votre compartiment Amazon S3.

Pour créer un filtre d'abonnements pour Amazon S3

1. Créez un compartiment Amazon Simple Storage Service (Amazon S3). Nous vous recommandons d'utiliser un compartiment qui a été créé spécifiquement pour CloudWatch Logs. Toutefois, si vous souhaitez utiliser un compartiment existant, passez directement à l'étape 2.

   Exécutez la commande suivante en remplaçant l'espace réservé à la région par la région que vous voulez utiliser :

   ```
   aws s3api create-bucket --bucket amzn-s3-demo-bucket2 --create-bucket-configuration
    LocationConstraint=region
   ```

   > **Note**
   >
   > `amzn-s3-demo-bucket2` est un exemple de nom de compartiment Amazon S3. Il est réservé. Pour que cette procédure fonctionne, vous devez le remplacer par le nom unique de votre compartiment Amazon S3.

   Voici un exemple de sortie :

   ```
   {
       "Location": "/amzn-s3-demo-bucket2"
   }
   ```

2. Créez le rôle IAM qui accorde à CloudWatch Logs l'autorisation de placer des données dans votre compartiment Amazon S3. Cette politique comprend une clé de contexte de condition aws:SourceArn globale pour aider à prévenir le problème de sécurité du député confus. Pour plus d'informations consultez Prévention du problème de l'adjoint confus.

   a. Utilisez un éditeur de texte pour créer une politique d'approbation dans un fichier `~/TrustPolicyForCWL.json` comme suit :

```
{
    "Statement": {
        "Effect": "Allow",
        "Principal": { "Service": "logs.amazonaws.com" },
        "Condition": {
            "StringLike": {
                "aws:SourceArn": "arn:aws:logs:region:123456789012:*"
            }
         },
        "Action": "sts:AssumeRole"
    }
}
```

b. Utilisez la commande create-role pour créer le rôle IAM, en spécifiant le fichier de politique d'approbation. Notez la valeur retournée de Role.Arn, car vous en aurez besoin ultérieurement :

```
aws iam create-role \
 --role-name CWLtoS3Role \
 --assume-role-policy-document file://~/TrustPolicyForCWL.json
{
    "Role": {
        "AssumeRolePolicyDocument": {
            "Statement": {
                "Action": "sts:AssumeRole",
                "Effect": "Allow",
                "Principal": {
                    "Service": "logs.amazonaws.com"
                },
                "Condition": {
                    "StringLike": {
                        "aws:SourceArn": "arn:aws:logs:region:123456789012:*"
                    }
                }
            }
        },
        "RoleId": "AAOIIAH450GAB4HC5F431",
        "CreateDate": "2015-05-29T13:46:29.431Z",
        "RoleName": "CWLtoS3Role",
        "Path": "/",
        "Arn": "arn:aws:iam::123456789012:role/CWLtoS3Role"
    }
}
```

3. Créez une politique d'autorisations pour définir les actions permises à CloudWatch Logs sur votre compte. D'abord, utilisez un éditeur de texte pour créer une stratégie d'autorisations dans un fichier `~/PermissionsForCWL.json`:

```
{
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["s3:PutObject"],
      "Resource": ["arn:aws:s3:::amzn-s3-demo-bucket2/*"]
    }
  ]
}
```

   Associez la politique d'autorisations au rôle à la commande `put-role-policy` suivante :

```
aws iam put-role-policy --role-name CWLtoS3Role --policy-name Permissions-Policy-For-
S3 --policy-document file://~/PermissionsForCWL.json
```

4. Créez un groupe de journaux `Delivery` ou utilisez un groupe de journaux `Delivery` existant.

```
aws logs create-log-group --log-group-name my-logs --log-group-class DELIVERY --
region REGION_NAME
```

5. `PutSubscriptionFilter` pour configurer la destination

```
aws logs put-subscription-filter
--log-group-name my-logs
--filter-name my-lambda-delivery
--filter-pattern ""
--destination-arn arn:aws:s3:::amzn-s3-demo-bucket2
--role-arn arn:aws:iam::123456789012:role/CWLtoS3Role
--region REGION_NAME
```

Envoi de journaux de fonction Lambda à Amazon S3

Dans la console Lambda, vous pouvez envoyer des journaux de fonction directement à Amazon S3 après avoir créé une nouvelle fonction. Pour ce faire, effectuez ces étapes :

1. Connectez-vous à Gestion de la console AWS et ouvrez la console Lambda.

2. Choisissez le nom de votre fonction.
3. Cliquez sur l'onglet Configuration.
4. Choisissez l'onglet Outils de surveillance et d'exploitation.
5. Dans le volet « Configuration de la journalisation », choisissez Modifier.
6. Dans la section « Contenu du journal », sélectionnez un format de journal.
7. Dans la section « Destination du journal » procédez comme suit :
   a. Sélectionnez un service de destination.
   b. Choisissez Créer un nouveau groupe de journaux ou utilisez un Groupe de journaux existant.

   > **Note**
   >
   > Si vous choisissez un groupe de journaux existant pour une destination Amazon S3, assurez-vous que le groupe de journaux que vous choisissez est un type de groupe de journaux `Delivery`.

   c. Choisissez un compartiment Amazon S3 comme destination pour vos journaux de fonction.
   d. Le groupe de journaux `Delivery` CloudWatch s'affichera.
8. Choisissez Enregistrer.

> **Note**
>
> Si le rôle IAM fourni dans la console ne dispose pas des autorisations requises, la configuration de la destination échouera. Pour résoudre ce problème, reportez-vous à la section Autorisations requises pour la destination du journal Amazon S3.

Journalisation entre comptes

Vous pouvez configurer Lambda pour envoyer des journaux vers un compartiment Amazon S3 dans un autre compte AWS. Cela nécessite de définir une destination et de configurer les autorisations appropriées dans les deux comptes.

Pour obtenir des instructions détaillées sur la configuration de la journalisation entre comptes, y compris les rôles et politiques IAM requis, consultez Configuration d'un nouvel abonnement entre comptes dans la documentation CloudWatch Logs.

Journalisation des appels d’API AWS Lambda avec AWS CloudTrail

AWS Lambda est intégré avec AWS CloudTrail, un service qui fournit un registre des actions prises par un utilisateur, un rôle ou un Service AWS. CloudTrail capture les appels d’API pour Lambda en tant qu’événements. Les appels capturés incluent des appels de la console Lambda et les appels de code vers les opérations d’API Lambda. À l’aide des informations collectées par CloudTrail, vous pouvez déterminer la demande qui a été faite à Lambda, l’adresse IP à partir de laquelle la demande a été faite, le moment où elle a été faite, ainsi que des détails supplémentaires.

Chaque événement ou entrée de journal contient des informations sur la personne ayant initié la demande. Les informations relatives à l’identité permettent de déterminer :

- Si la demande a été effectuée avec des informations d’identification d’utilisateur root ou d’utilisateur root.
- Si la demande a été faite au nom d'un utilisateur du centre d'identité IAM.
- Si la demande a été effectuée avec les informations d’identification de sécurité temporaires d’un rôle ou d’un utilisateur fédéré.
- Si la requête a été effectuée par un autre Service AWS.

CloudTrail est actif dans votre Compte AWS lorsque vous créez le compte et vous avez automatiquement accès à l'historique des événements CloudTrail. L'historique des événements de CloudTrail permet de visualiser, de rechercher, de télécharger et d'enregistrer de façon immuable les événements de gestion enregistrés au cours des 90 derniers jours dans un Région AWS. Pour plus d'informations, consultez Travailler avec l'historique des événements CloudTrail dans le AWS CloudTrailGuide de l'utilisateur. La consultation de l’historique des événements ne génère aucuns frais CloudTrail.

Pour un enregistrement permanent des événements dans vos Compte AWS 90 derniers jours, créez un historique ou un stockage de données d'événements CloudTrail Lake.

Journaux de suivi CloudTrail

> Un journal de suivi permet à CloudTrail de livrer des fichiers journaux à compartiment Amazon S3. Tous les journaux de suivi créés à l'aide de la AWS Management Console sont multi-régions. Vous ne pouvez créer un journal de suivi en une ou plusieurs régions à l'aide de l'AWS CLI. La création d'un journal de suivi multi-régions est recommandée, car vous pouvez journaliser l'activité

dans toutes Régions AWS dans votre compte. Si vous créez un journal de suivi pour une seule région, il convient de n'afficher que les événements enregistrés dans le journal de suivi pour une seule région Région AWS. Pour plus d'informations sur les journaux de suivi, consultez Créez un journal de suivi dans vos Compte AWS et Création d'un journal de suivi pour une organisation dans le AWS CloudTrail Guide de l'utilisateur.

Vous pouvez diffuser une copie de vos événements de gestion en cours à votre compartiment Amazon S3 sans frais depuis CloudTrail en créant un suivi. Toutefois, des frais de stockage Amazon S3 sont facturés. Pour en savoir plus sur la tarification CloudTrail, consultez Tarification d'AWS CloudTrail. Pour obtenir des informations sur la tarification Amazon S3, consultez Tarification Amazon S3.

Magasins de données d'événement CloudTrail Lake

CloudTrail Lake vous permet d'exécuter des requêtes basées sur SQL sur vos événements. CloudTrail Lake convertit les événements existants au format JSON basé sur des lignes au format Apache ORC. ORC est un format de stockage en colonnes qui est optimisé pour une récupération rapide des données. Les événements sont agrégés dans des magasins de données d'événement. Ceux-ci constituent des collections immuables d'événements basées sur des critères que vous sélectionnez en appliquant des sélecteurs d'événements avancés. Les sélecteurs que vous appliquez à un magasin de données d'événement contrôlent les événements qui persistent et que vous pouvez interroger. Pour plus d'informations sur CloudTrail Lake, consultez Utilisation de AWS CloudTrail Lake dans le AWS CloudTrail Guide de l'utilisateur.

Le stockage des données d'événements CloudTrail Lake et les requêtes entraînent des coûts. Lorsque vous créez un magasin de données d'événement, vous choisissez l'option de tarification que vous voulez utiliser pour le magasin de données d'événement. L'option de tarification détermine le coût d'ingestion et de stockage des événements, ainsi que les périodes de conservation par défaut et maximale pour le magasin de données d'événement. Pour en savoir plus sur la tarification CloudTrail, consultez Tarification d'AWS CloudTrail.

Événements de données Lambda dans CloudTrail

Les événements de données fournissent des informations sur les opérations de ressources effectuées sur ou dans une ressource (par exemple, lecture ou écriture de données dans un objet Amazon S3). Ils sont également connus sous le nom opérations de plans de données. Les événements de données sont souvent des activités dont le volume est élevé. Par défaut, CloudTrail

ne journalise pas la plupart des événements de données, et l'Historique des événements CloudTrail ne les enregistre pas.

L'un des événements de données CloudTrail qui est journalisé par défaut pour les services pris en charge est `LambdaESMDisabled`. Pour en savoir plus sur l'utilisation de cet événement afin de résoudre les problèmes liés aux mappages des sources d'événements Lambda, consultez the section called "Utilisation de CloudTrail pour dépanner les sources d'événement Lambda désactivées".

Des frais supplémentaires s'appliquent pour les événements de données. Pour en savoir plus sur la tarification CloudTrail, consultez Tarification d'AWS CloudTrail.

Vous pouvez journaliser les événements de données pour le type de ressources `AWS::Lambda::Function` en utilisant la console CloudTrail, l'AWS CLI ou les opérations d'API CloudTrail. Pour plus d'informations sur la façon de journaliser les événements de données, consultez Journalisation des événements de données avec la AWS Management Console et Journalisation des événements de données avec l'AWS Command Line Interface dans le Guide de l'utilisateur AWS CloudTrail.

Le tableau suivant répertorie les types de ressources Lambda pour lesquels vous pouvez journaliser les événements de données. La colonne Type d'événement de données (console) indique la valeur à choisir dans la liste Type d'événement de données de la console CloudTrail. La colonne resources.type value indique la valeur de `resources.type`, que vous devez spécifier lors de la configuration des sélecteurs d'événements avancés à l'aide de l'AWS CLI ou des API CloudTrail. La colonne API de données journalisées dans CloudTrail indique les appels d'API journalisés dans CloudTrail pour le type de ressource.

Type d'événement de données (console)	valeur resources.type	API de données journalisées dans CloudTrail
Lambda	`AWS::Lambda::Function`	Invocation

Vous pouvez configurer des sélecteurs d'événements avancés pour filtrer les champs `eventName`, `readOnly` et `resources.ARN` afin de ne journaliser que les événements importants pour vous. L'exemple suivant est la vue JSON d'une configuration d'événements de données qui journalise les événements pour une fonction spécifique uniquement. Pour plus d'informations sur ces champs, consultez AdvancedFieldSelector dans la Référence d'API AWS CloudTrail.

```
[
  {
    "name": "function-invokes",
    "fieldSelectors": [
      {
        "field": "eventCategory",
        "equals": [
          "Data"
        ]
      },
      {
        "field": "resources.type",
        "equals": [
          "AWS::Lambda::Function"
        ]
      },
      {
        "field": "resources.ARN",
        "equals": [
          "arn:aws:lambda:us-east-1:111122223333:function:hello-world"
        ]
      }
    ]
  }
]
```

Événements de gestion Lambda dans CloudTrail

Les événements de gestion fournissent des informations sur les opérations de gestion exécutées sur des ressources de votre Compte AWS. Ils sont également connus sous le nom d'opérations de plan de contrôle. Par défaut, CloudTrail journalise les événements de gestion.

Lambda prend en charge la journalisation des actions suivantes en tant qu'événements de gestion dans les fichiers journaux CloudTrail.

> **Note**
>
> Dans le fichier journal CloudTrail, `eventName` peut inclure des informations de date et de version, mais il s'agit toujours de la même action d'API publique. Par exemple, l'action `GetFunction` apparaît sous la forme de `GetFunction20150331v2`. La liste suivante indique les cas où le nom de l'événement diffère du nom de l'action d'API.

- AddLayerVersionPermission
- AddPermission (nom de l'événement : `AddPermission20150331v2`)
- CreateAlias (nom de l'événement : `CreateAlias20150331`)
- CreateEventSourceMapping (nom de l'événement : `CreateEventSourceMapping20150331`)
- CreateFunction (nom de l'événement : `CreateFunction20150331`)

  (Les paramètres `Environment` et `ZipFile` ne figurent pas dans les journaux CloudTrail pour `CreateFunction`.)
- CreateFunctionUrlConfig
- DeleteAlias (nom de l'événement : `DeleteAlias20150331`)
- DeleteCodeSigningConfig
- DeleteEventSourceMapping (nom de l'événement : `DeleteEventSourceMapping20150331`)
- DeleteFunction (nom de l'événement : `DeleteFunction20150331`)
- DeleteFunctionConcurrency (nom de l'événement : `DeleteFunctionConcurrency20171031`)
- DeleteFunctionUrlConfig
- DeleteProvisionedConcurrencyConfig
- GetAlias (nom de l'événement : `GetAlias20150331`)
- GetEventSourceMapping
- GetFunction
- GetFunctionUrlConfig
- GetFunctionConfiguration
- GetLayerVersionPolicy
- GetPolicy
- ListEventSourceMappings
- ListFunctions
- ListFunctionUrlConfigs
- PublishLayerVersion (nom de l'événement : `PublishLayerVersion20181031`)

  (Le paramètre `ZipFile` ne figure pas dans les journaux CloudTrail pour `PublishLayerVersion`.)

- PublishVersion (nom de l'événement : `PublishVersion20150331`)
- PutFunctionConcurrency (nom de l'événement : `PutFunctionConcurrency20171031`)
- PutFunctionCoDesigningConfig
- PutFunctionEventInvokeConfig
- PutProvisionedConcurrencyConfig
- PutRuntimeManagementConfig
- RemovePermission (nom de l'événement : `RemovePermission20150331v2`)
- TagResource (nom de l'événement : `TagResource20170331v2`)
- UntagResource (nom de l'événement : `UntagResource20170331v2`)
- UpdateAlias (nom de l'événement : `UpdateAlias20150331`)
- UpdateCodeSigningConfig
- UpdateEventSourceMapping (nom de l'événement : `UpdateEventSourceMapping20150331`)
- UpdateFunctionCode (nom de l'événement : `UpdateFunctionCode20150331v2`)

  (Le paramètre `ZipFile` ne figure pas dans les journaux CloudTrail pour `UpdateFunctionCode`.)
- UpdateFunctionConfiguration (nom de l'événement : `UpdateFunctionConfiguration20150331v2`)

  (Le paramètre `Environment` ne figure pas dans les journaux CloudTrail pour `UpdateFunctionConfiguration`.)
- UpdateFunctionEventInvokeConfig
- UpdateFunctionUrlConfig

Utilisation de CloudTrail pour dépanner les sources d'événement Lambda désactivées

Lorsque vous modifiez l'état d'un mappage des sources d'événements à l'aide de l'action d'API UpdateEventSourceMapping, l'appel d'API est journalisé en tant qu'événement de gestion dans CloudTrail. Les mappages de sources d'événements peuvent également passer directement à l'état `Disabled` en raison d'erreurs.

Pour les services suivants, Lambda publie l'événement de données `LambdaESMDisabled` sur CloudTrail lorsque la source de l'événement passe à l'état Disabled :

- Amazon Simple Queue Service (Amazon SQS)
- Amazon DynamoDB
- Amazon Kinesis

Lambda ne prend en charge cet événement pour aucun autre type de mappage des sources d'événements.

Pour recevoir des alertes lorsque les mappages des sources d'événements pour les services pris en charge passent à l'état `Disabled`, configurez une alarme dans Amazon CloudWatch à l'aide de l'événement CloudTrail `LambdaESMDisabled`. Pour plus d'informations sur la configuration d'une alarme CloudWatch, consultez la rubrique Creating CloudWatch alarms for CloudTrail events: examples.

L'entité `serviceEventDetails` figurant dans le message d'événement `LambdaESMDisabled` contient l'un des codes d'erreur suivants.

`RESOURCE_NOT_FOUND`

La ressource spécifiée dans la demande n'existe pas.

`FUNCTION_NOT_FOUND`

La fonction attachée à la source d'événement n'existe pas.

`REGION_NAME_NOT_VALID`

Un nom de région fourni à la source ou à la fonction d'événement n'est pas valide.

`AUTHORIZATION_ERROR`

Les autorisations n'ont pas été définies ou sont mal configurées.

`FUNCTION_IN_FAILED_STATE`

Le code de fonction ne compile pas, a rencontré une exception irrécupérable ou un mauvais déploiement s'est produit.

Exemples d'événements Lambda

Un événement représente une demande d'une source quelconque et comprend des informations sur l'opération d'API demandée, y compris la date et l'heure de l'opération, les paramètres de la

demande, etc. Les fichiers journaux CloudTrail ne constituent pas une série ordonnée des appels d'API publics. Les événements ne suivent aucun ordre précis.

L'exemple suivant montre les entrées de journal CloudTrail pour les actions `GetFunction` et `DeleteFunction`.

> **Note**
>
> `eventName` peut inclure des informations de date et de version, comme `"GetFunction20150331"`, mais il s'agit toujours de la même API publique.

```
{
  "Records": [
    {
      "eventVersion": "1.03",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "A1B2C3D4E5F6G7EXAMPLE",
        "arn": "arn:aws:iam::111122223333:user/myUserName",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "myUserName"
      },
      "eventTime": "2015-03-18T19:03:36Z",
      "eventSource": "lambda.amazonaws.com",
      "eventName": "GetFunction",
      "awsRegion": "us-east-1",
      "sourceIPAddress": "127.0.0.1",
      "userAgent": "Python-httplib2/0.8 (gzip)",
      "errorCode": "AccessDenied",
      "errorMessage": "User: arn:aws:iam::111122223333:user/myUserName is not
 authorized to perform: lambda:GetFunction on resource: arn:aws:lambda:us-
west-2:111122223333:function:other-acct-function",
      "requestParameters": null,
      "responseElements": null,
      "requestID": "7aebcd0f-cda1-11e4-aaa2-e356da31e4ff",
      "eventID": "e92a3e85-8ecd-4d23-8074-843aabfe89bf",
      "eventType": "AwsApiCall",
      "recipientAccountId": "111122223333"
    },
    {
```

```
      "eventVersion": "1.03",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "A1B2C3D4E5F6G7EXAMPLE",
        "arn": "arn:aws:iam::111122223333:user/myUserName",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "myUserName"
      },
      "eventTime": "2015-03-18T19:04:42Z",
      "eventSource": "lambda.amazonaws.com",
      "eventName": "DeleteFunction20150331",
      "awsRegion": "us-east-1",
      "sourceIPAddress": "127.0.0.1",
      "userAgent": "Python-httplib2/0.8 (gzip)",
      "requestParameters": {
        "functionName": "basic-node-task"
      },
      "responseElements": null,
      "requestID": "a2198ecc-cda1-11e4-aaa2-e356da31e4ff",
      "eventID": "20b84ce5-730f-482e-b2b2-e8fcc87ceb22",
      "eventType": "AwsApiCall",
      "recipientAccountId": "111122223333"
    }
  ]
}
```


Pour plus d'informations sur le contenu des enregistrements CloudTrail, consultez CloudTrail record contents dans le Guide de l'utilisateur AWS CloudTrail.

Visualisez les invocations de fonctions Lambda à l'aide de AWS X-Ray

Vous pouvez l'utiliser AWS X-Ray pour visualiser les composants de votre application, identifier les goulots d'étranglement liés aux performances et résoudre les demandes ayant entraîné une erreur. Vos fonctions Lambda envoient des données de suivi à X-Ray qui les traite pour générer une cartographie de service et des résumés de suivi pouvant faire l'objet d'une recherche.

Lambda prend en charge deux modes de suivi pour X-Ray : `Active` et `PassThrough`. Grâce au suivi `Active`, Lambda crée automatiquement des segments de suivi pour les invocations de fonction et les envoie à X-Ray. Le mode `PassThrough`, quant à lui, propage simplement le contexte de suivi aux services en aval. Si vous avez activé le suivi `Active` pour votre fonction, Lambda envoie automatiquement les suivis à X-Ray pour les demandes échantillonnées. Généralement, un service en amont, tel qu'Amazon API Gateway ou une application hébergée sur Amazon EC2 équipée du SDK X-Ray, décide si les demandes entrantes doivent être suivies, puis ajoute cette décision d'échantillonnage en tant qu'en-tête de suivi. Lambda utilise cet en-tête pour décider d'envoyer des traces ou non. Les traces provenant des producteurs de messages en amont, tels qu'Amazon SQS, sont automatiquement liées aux traces des fonctions Lambda en aval, créant ainsi une end-to-end vue de l'ensemble de l'application. Pour plus d'informations, consultez Traçage des applications événementielles dans le Guide du développeur AWS X-Ray .

> **Note**
>
> Le suivi X-Ray n'est actuellement pas pris en charge pour les fonctions Lambda avec Amazon Managed Streaming for Apache Kafka (Amazon MSK), Apache Kafka autogéré, Amazon MQ avec ActiveMQ et RabbitMQ ou les mappages des sources d'événements Amazon DocumentDB.

Pour activer/désactiver le traçage actif sur votre fonction Lambda avec la console, procédez comme suit :

Pour activer le traçage actif

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Choisissez Configuration, puis choisissez Outils de surveillance et d'opérations.

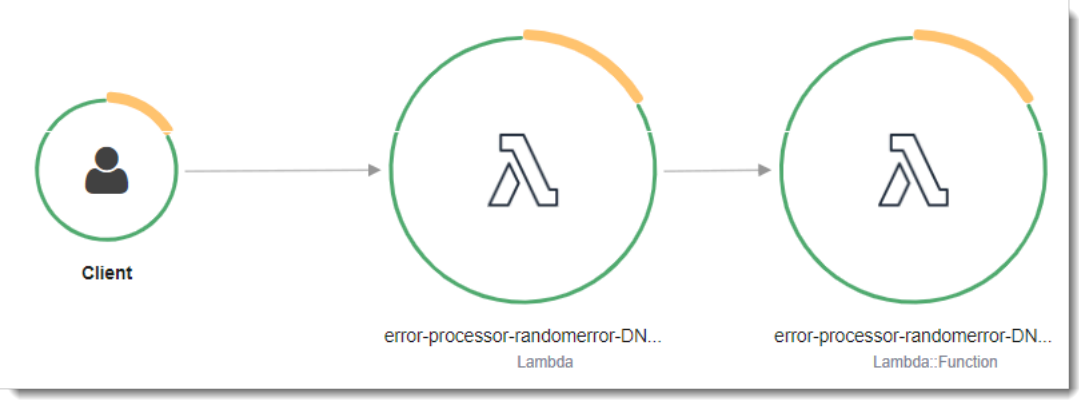
4. Dans Outils de surveillance supplémentaires, choisissez Modifier.
5. Sous Signaux CloudWatch d'application et AWS X-Ray sélectionnez Activer les traces de service Lambda.
6. Choisissez Enregistrer.

Votre fonction a besoin d'une autorisation pour charger des données de suivi vers X-Ray. Lorsque vous activez le suivi actif dans la console Lambda, Lambda ajoute les autorisations requises au rôle d'exécution de votre fonction. Dans le cas contraire, ajoutez la AWSXRayDaemonWriteAccesspolitique au rôle d'exécution.

X-Ray ne trace pas toutes les requêtes vers votre application. X-Ray applique un algorithme d'échantillonnage pour s'assurer que le suivi est efficace, tout en fournissant un échantillon représentatif de toutes les demandes. Le taux d'échantillonnage est 1 demande par seconde et 5 % de demandes supplémentaires. Vous ne pouvez pas configurer ce taux d'échantillonnage X-Ray pour vos fonctions.

Comprendre les suivis X-Ray

Dans X-Ray, un suivi enregistre des informations sur une demande traitée par un ou plusieurs services. Lambda enregistre deux segments par suivi, ce qui a pour effet de créer deux nœuds sur le graphique du service. L'image suivante met en évidence ces deux nœuds :



Le premier nœud sur la gauche représente le service Lambda qui reçoit la demande d'invocation. Le deuxième nœud représente votre fonction Lambda spécifique.

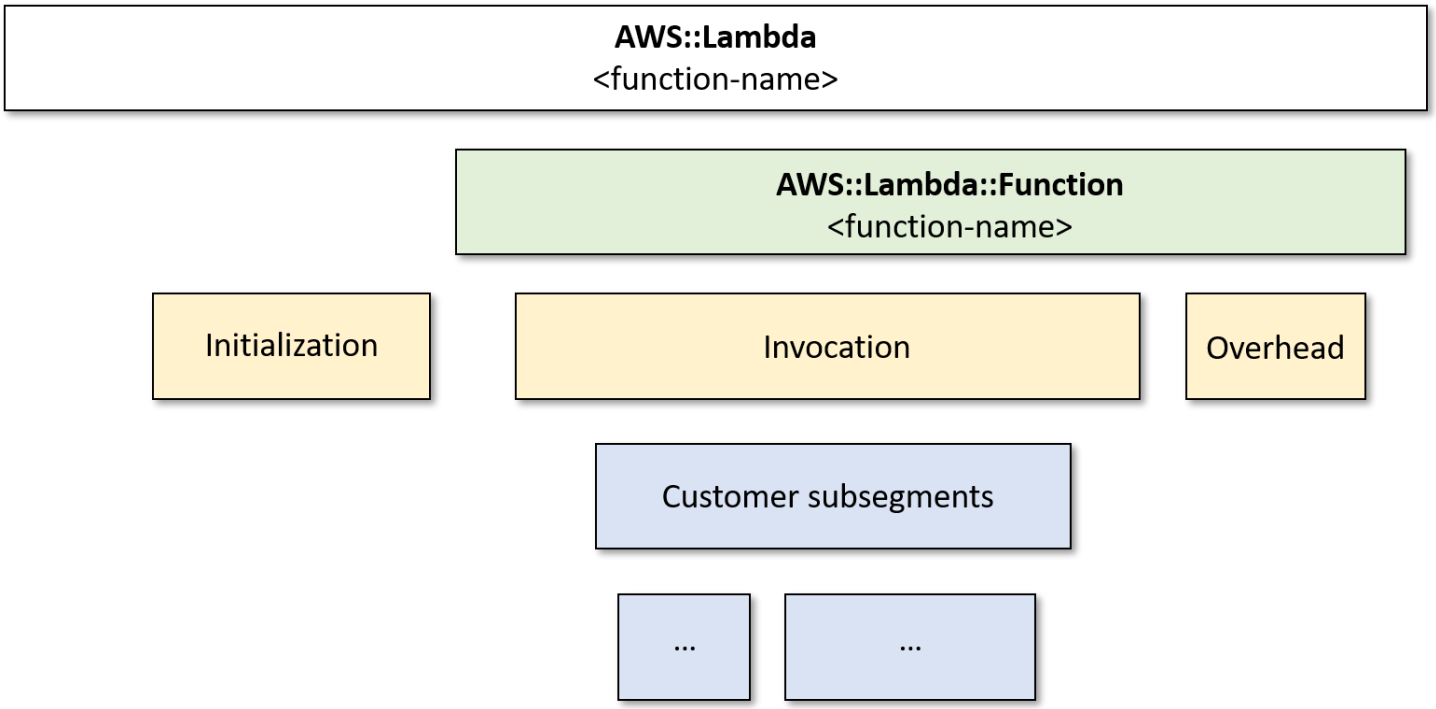
Le segment enregistré pour le service Lambda, `AWS::Lambda`, couvre toutes les étapes nécessaires à la préparation de l'environnement d'exécution Lambda. Cela inclut la planification de la MicroVM, la création ou le déblocage d'un environnement d'exécution avec les ressources que vous avez configurées, ainsi que le téléchargement de votre code de la fonction et de toutes les couches.

Le segment `AWS::Lambda::Function` concerne le travail effectué par la fonction.

> **Note**
>
> AWS met actuellement en œuvre des modifications du service Lambda. En raison de ces modifications, vous pouvez constater des différences mineures entre la structure et le contenu des messages du journal système et des segments de suivi émis par les différentes fonctions Lambda de votre Compte AWS.
> Cette modification affecte les sous-segments du segment de fonction. Les paragraphes suivants décrivent les anciens et les nouveaux formats de ces sous-segments.
> Ces modifications seront mises en œuvre au cours des prochaines semaines, et toutes les fonctions, Régions AWS sauf en Chine et dans les GovCloud régions, seront transférées pour utiliser le nouveau format des messages de journal et des segments de trace.

Structure de segment AWS X-Ray Lambda à l'ancienne

L'ancienne structure X-Ray de segment `AWS::Lambda` ressemble à ce qui suit :



Dans ce format, le segment de fonction comporte des sous-segments pour `Initialization`, `Invocation` et `Overhead`. Pour Lambda SnapStart uniquement, il existe également un sous-segment `Restore` (non représenté sur ce schéma).

Le sous-segment `Initialization` représente la phase d'initialisation du cycle de vie de l'environnement d'exécution Lambda. Au cours de cette phase, Lambda initialise les extensions, initialise l'environnement d'exécution, et exécute le code d'initialisation de la fonction.

Le sous-segment `Invocation` représente la phase d'invocation où Lambda invoque le gestionnaire de fonction. Cela commence par l'enregistrement de l'exécution et de l'extension et se termine lorsque l'exécution est prête à envoyer la réponse.

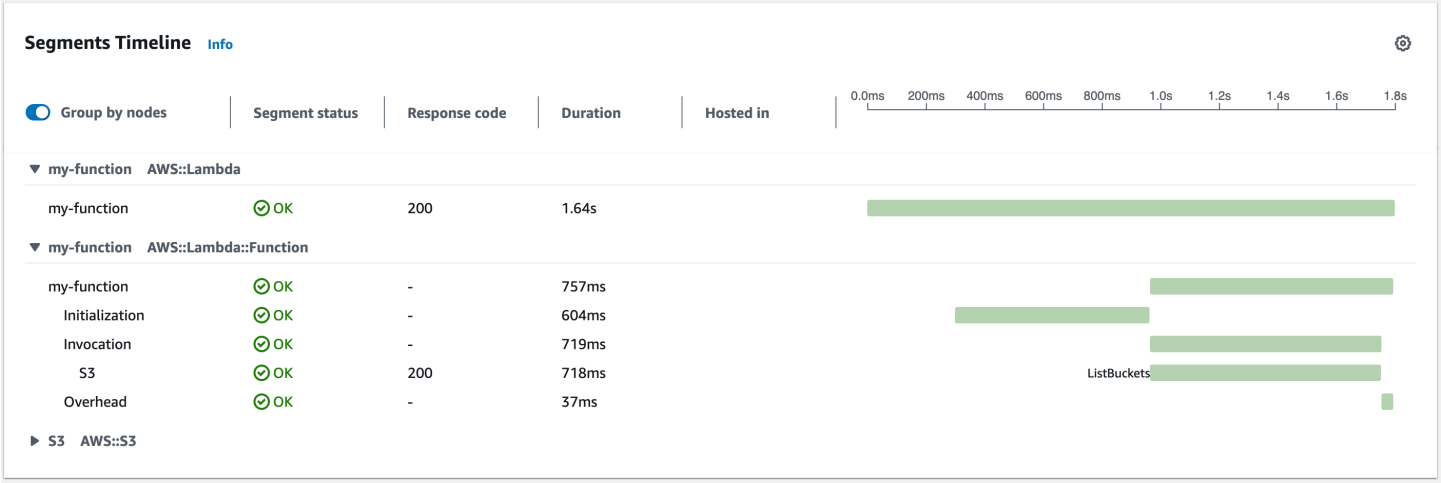
(Lambda SnapStart uniquement) Le `Restore` sous-segment indique le temps nécessaire à Lambda pour restaurer un instantané, charger le moteur d'exécution et exécuter les éventuels hooks d'exécution après la restauration. Le processus de restauration des instantanés peut inclure du temps consacré à des activités en dehors de la MicroVM. Cette heure est indiquée dans le sous-segment `Restore`. Le temps passé en dehors de la microVM pour restaurer un instantané ne vous est pas facturé.

Le sous-segment `Overhead` représente la phase qui se produit entre le moment où l'exécution envoie la réponse et le signal pour l'invocation suivante. Pendant ce temps, l'exécution termine toutes les tâches liées à une invocation et se prépare à geler l'environnement de test (sandbox).

> **Important**
>
> Vous pouvez utiliser le kit SDK X-Ray pour étendre le sous-segment `Invocation` avec des sous-segments supplémentaires pour les appels, les annotations et les métadonnées en aval. Vous ne pouvez pas accéder directement au segment de fonction ou enregistrer une tâche effectuée en dehors de la portée d'invocation du gestionnaire.

Pour plus d'informations sur les phases de l'environnement d'exécution Lambda, consultez the section called "Environnement d'exécution".

Un exemple de suivi utilisant l'ancienne structure X-Ray est illustré dans le schéma suivant.

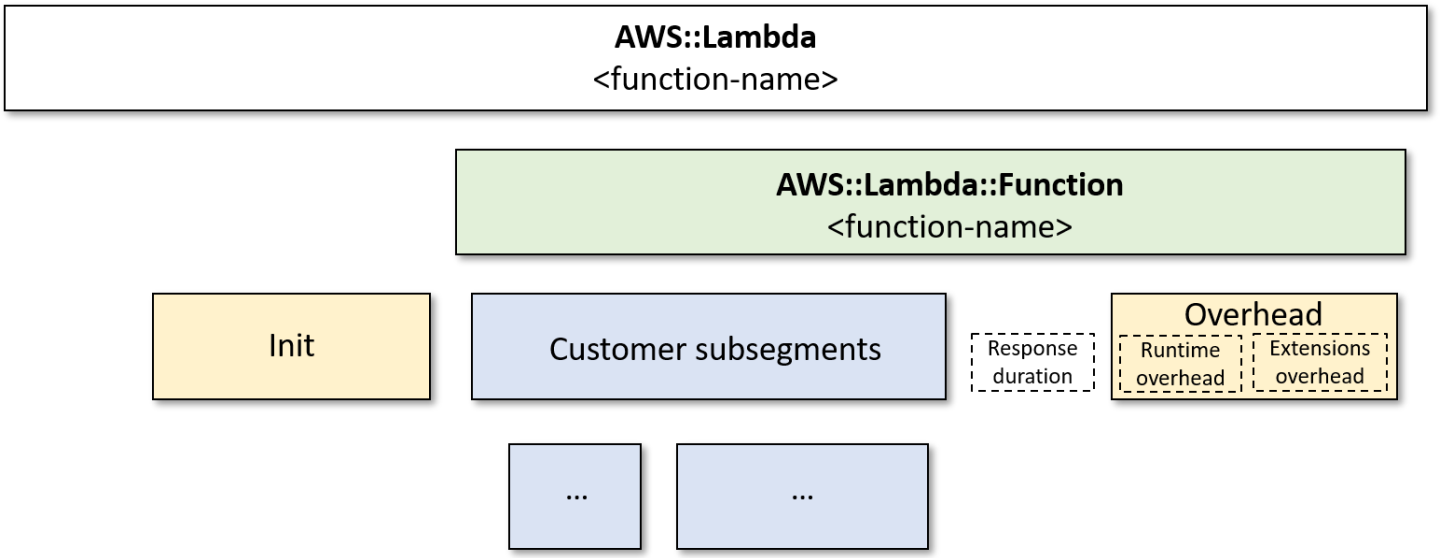


Notez les deux segments de l'exemple. Les deux sont nommés my-function, mais l'un a pour origine `AWS::Lambda` et l'autre a pour origine `AWS::Lambda::Function`. Si le segment `AWS::Lambda` affiche une erreur, cela signifie que le service Lambda a rencontré un problème. Si le segment `AWS::Lambda::Function` affiche une erreur, cela signifie que votre fonction a rencontré un problème.

> **Note**
>
> Occasionnellement, vous pouvez remarquer un grand écart entre les phases d'initialisation et d'invocation de la fonction dans vos traces X-Ray. Pour les fonctions utilisant la simultanéité provisionnée, cela est dû au fait que Lambda initialise vos instances de fonction bien avant l'invocation. Pour les fonctions utilisant la simultanéité non réservée (à la demande), Lambda peut initialiser de manière proactive une instance de fonction, même s'il n'y a pas d'invocation. Visuellement, ces deux cas se manifestent par un écart de temps entre les phases d'initialisation et d'invocation.

Structure de segment AWS X-Ray Lambda de style nouveau

La nouvelle structure X-Ray de segment `AWS::Lambda` ressemble à ce qui suit :

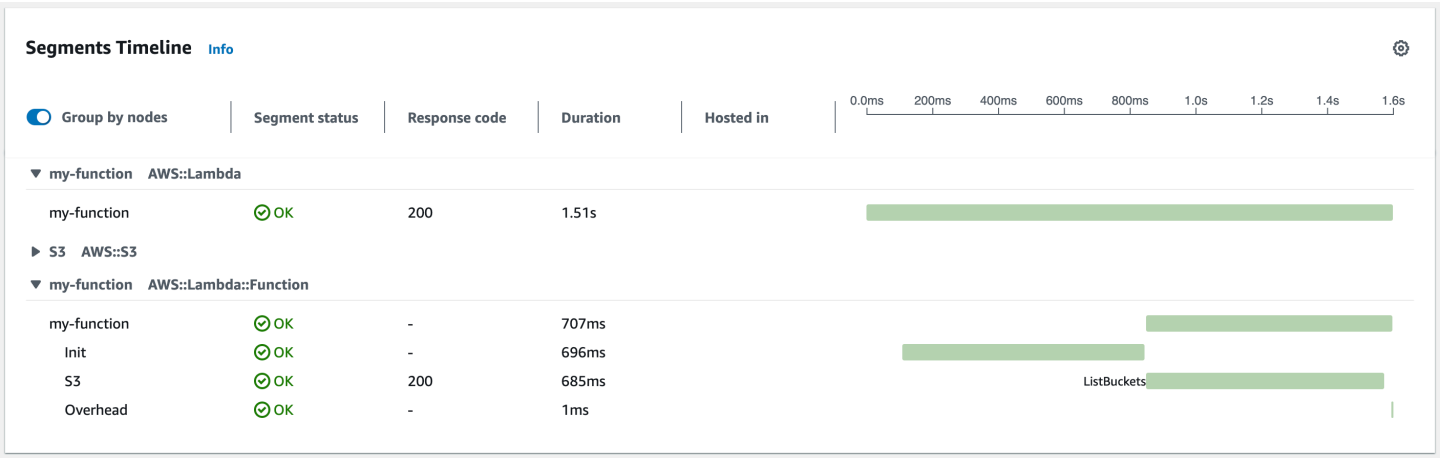


Dans ce nouveau format, le sous-segment `Init` représente la phase d'initialisation du cycle de vie de l'environnement d'exécution Lambda comme auparavant.

Le nouveau format ne contient pas de segment d'invocation. À la place, les sous-segments du client sont directement rattachés au segment `AWS::Lambda::Function`. Ce segment contient les métriques suivantes sous forme d'annotations :

- `aws.responseLatency` : le temps nécessaire à l'exécution de la fonction
- `aws.responseDuration` : le temps nécessaire pour transmettre la réponse au client
- `aws.runtimeOverhead` : le temps supplémentaire dont l'exécution a eu besoin pour se terminer
- `aws.extensionOverhead` : le temps supplémentaire dont les extensions ont eu besoin pour se terminer

Un exemple de suivi utilisant la nouvelle structure X-Ray est illustré dans le schéma suivant.



Notez les deux segments de l'exemple. Les deux sont nommés my-function, mais l'un a pour origine `AWS::Lambda` et l'autre a pour origine `AWS::Lambda::Function`. Si le segment `AWS::Lambda` affiche une erreur, cela signifie que le service Lambda a rencontré un problème. Si le segment `AWS::Lambda::Function` affiche une erreur, cela signifie que votre fonction a rencontré un problème.

Consultez les rubriques suivantes pour une présentation du suivi dans Lambda spécifique de chaque langage :

- Instrumentation du code Node.js dans AWS Lambda
- Instrumentation du code Python dans AWS Lambda
- Instrumentation du code Ruby dans AWS Lambda
- Instrumentation du code Java dans AWS Lambda
- Instrumentation du code Go dans AWS Lambda
- Instrumentation du code C # dans AWS Lambda

Pour obtenir la liste complète des services qui prennent en charge l'instrumentation active, consultez Services AWS pris en charge dans le Guide du développeur AWS X-Ray .

Comportement de suivi par défaut dans Lambda

Si le suivi `Active` n'est pas activé, Lambda passe en mode de suivi `PassThrough` par défaut.

En mode `PassThrough`, Lambda transmet l'en-tête de suivi X-Ray aux services en aval, mais n'envoie pas de suivis automatiquement. Cela est vrai même si l'en-tête de suivi contient la décision d'échantillonner la demande. Si le service en amont ne fournit pas d'en-tête de suivi X-Ray, Lambda génère un en-tête et prend la décision de ne pas échantillonner. Cependant, vous pouvez envoyer vos propres suivis en appelant des bibliothèques de suivi à partir du code de votre fonction.

> **Note**
>
> Auparavant, Lambda envoyait automatiquement des traces lorsque des services en amont, comme Amazon API Gateway, ajoutaient un en-tête de suivi. En n'envoyant pas de traces automatiquement, Lambda vous permet de contrôler les fonctions qui sont importantes pour vous. Si votre solution repose sur ce comportement de suivi passif, passez au suivi `Active`.

Autorisations du rôle d'exécution

Lambda a besoin des autorisations suivantes pour envoyer des données de suivi à X-Ray. Ajoutez-les au rôle d'exécution de la fonction.

- radiographie : PutTraceSegments
- radiographie : PutTelemetryRecords

Ces autorisations sont incluses dans la politique AWSXRayDaemonWriteAccessgérée.

Activation du suivi **`Active`** avec l'API Lambda

Pour gérer la configuration du suivi avec le AWS SDK AWS CLI ou le SDK, utilisez les opérations d'API suivantes :

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

L'exemple de AWS CLI commande suivant active le suivi actif sur une fonction nommée my-function.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Le mode de suivi fait partie de la configuration spécifique de la version lorsque vous publiez une version de votre fonction. Vous ne pouvez pas modifier le mode de suivi sur une version publiée.

Activation du **`Active`** suivi avec CloudFormation

Pour activer le suivi d'une `AWS::Lambda::Function` ressource dans un CloudFormation modèle, utilisez la `TracingConfig` propriété.

Example function-inline.yml – Configuration du suivi

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
```

```
      TracingConfig:
        Mode: Active
      ...
```

Pour une `AWS::Serverless::Function` ressource AWS Serverless Application Model (AWS SAM), utilisez la `Tracing` propriété.

Example template.yml – Configuration du suivi

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

Surveillance des performances de fonctions avec Amazon CloudWatch Lambda Insights

Amazon CloudWatch Lambda Insights collecte et agrège les métriques et journaux de performances de runtime de fonction Lambda pour vos applications sans serveur. Cette page explique comment activer et utiliser Lambda Insights pour diagnostiquer des problèmes liés à vos fonctions Lambda.

Sections

- Comment Lambda Insights surveille les applications sans serveur
- Tarification
- Environnements d'exécution pris en charge
- Activation de Lambda Insights dans la console Lambda
- Activation de Lambda Insights par programme
- Utilisation du tableau de bord Lambda Insights
- Exemple de flux de travail permettant de détecter les anomalies de fonction
- Exemple de flux de travail utilisant des demandes pour dépanner une fonction
- Quelle est la prochaine étape ?

Comment Lambda Insights surveille les applications sans serveur

CloudWatch Lambda Insights est une solution de surveillance et de dépannage pour les applications sans serveur s'exécutant sur AWS Lambda. La solution collecte, agrège et résume les métriques de niveau système, notamment l'utilisation de temps d'UC, de mémoire, de disque et de réseau. Elle collecte, agrège et résume également des informations de diagnostic telles que des démarrages à froid et des arrêts de rôle de travail Lambda pour vous aider à circonscrire des problèmes liés à vos fonctions Lambda, ainsi qu'à les résoudre rapidement.

Lambda Insights utilise une nouvelle extension CloudWatch Lambda Insights, fournie sous la forme d'une couche Lambda. Lorsque vous activez cette extension sur une fonction Lambda pour une exécution prise en charge, elle collecte des métriques au niveau du système et émet un événement du journal des performances unique pour chaque invocation de cette fonction Lambda. CloudWatch utilise une mise en forme de métrique intégrée pour extraire les métriques des événements de journal. Pour de plus amples informations, veuillez consulter la section relative à l'utilisation des extensions AWS Lambda.

La couche Lambda Insights étend `CreateLogStream` et `PutLogEvents` pour le groupe de journaux `/aws/lambda-insights/`.

Tarification

Lorsque vous activez Lambda Insights pour votre fonction Lambda, Lambda Insights signale 8 métriques par fonction et chaque invocation de fonction envoie environ 1 Ko de données de journal à CloudWatch. Vous payez uniquement les métriques et les journaux signalés pour votre fonction par Lambda Insights. Aucun frais minimum ni aucune politique d'utilisation obligatoire des services ne sont appliqués. Vous ne payez pas pour Lambda Insights si la fonction n'est pas appelée. Pour un exemple de tarification, consultez Tarification Amazon CloudWatch.

Environnements d'exécution pris en charge

Vous pouvez utiliser Lambda Insights avec n'importe quel runtime prenant en charge les extensions Lambda.

Activation de Lambda Insights dans la console Lambda

Vous pouvez activer la surveillance améliorée de Lambda Insights sur des fonctions Lambda nouvelles et existantes. Lorsque vous activez Lambda Insights sur une fonction dans la console Lambda pour un runtime pris en charge, Lambda ajoute l'extension Lambda Insights en tant que couche à votre fonction, et vérifie ou tente d'attacher la stratégie `CloudWatchLambdaInsightsExecutionRolePolicy` au rôle d'exécution de votre fonction.

Pour activer Lambda Insights dans la console Lambda

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez votre fonction.
3. Choisissez l'onglet Configuration.
4. Dans le menu de gauche, sélectionnez Outils de surveillance et d’exploitation.
5. Dans le volet Outils de surveillance supplémentaires, choisissez Modifier.
6. Sous CloudWatch Lambda Insights, activez Surveillance améliorée.
7. Choisissez Enregistrer.

Activation de Lambda Insights par programme

Vous pouvez également activer Lambda Insights à l'aide de l'AWS Command Line Interface (AWS CLI), de l'interface de ligne de commande AWS Serverless Application Model (SAM), d'CloudFormation ou de l'AWS Cloud Development Kit (AWS CDK). Lorsque vous activez Lambda Insights par programmation sur une fonction pour un runtime pris en charge, CloudWatch attache la stratégie `CloudWatchLambdaInsightsExecutionRolePolicy` au rôle d'exécution de votre fonction.

Pour plus d'informations, consultez Mise en route avec Lambda Insights dans le Guide de l'utilisateur Amazon CloudWatch.

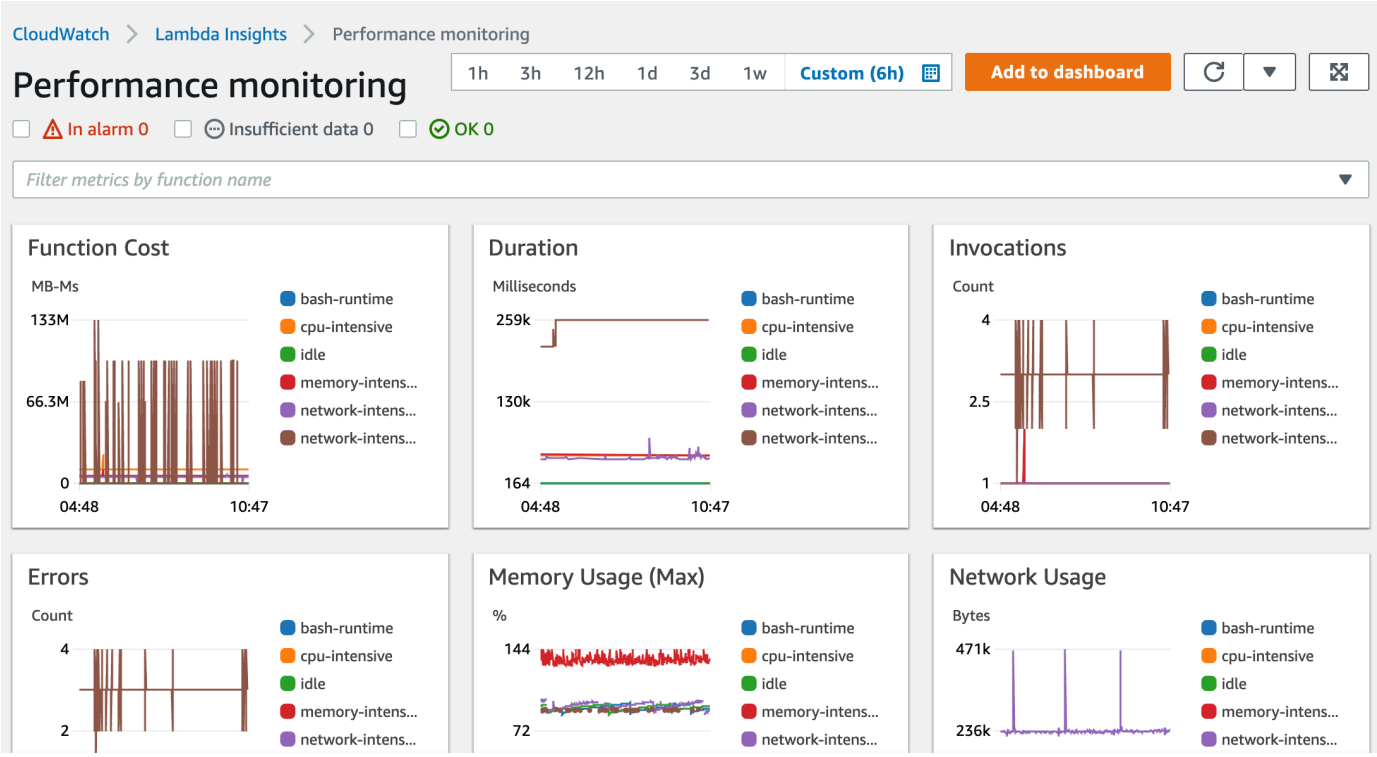
Utilisation du tableau de bord Lambda Insights

Le tableau de bord Lambda Insights a deux modes d'affichage dans la console CloudWatch : la vue d'ensemble multifonction et la vue de fonction unique. La vue d'ensemble multifonction regroupe les métriques de runtime pour les fonctions Lambda dans le compte AWS et la région actuels. La vue de fonction unique affiche les métriques de runtime disponibles pour une seule fonction Lambda.

Vous pouvez utiliser la vue d'ensemble multifonction du tableau de bord Lambda Insights dans la console CloudWatch pour identifier les fonctions Lambda sur-utilisées et sous-utilisées. Vous pouvez utiliser la vue de fonction unique du tableau de bord Lambda Insights dans la console CloudWatch pour résoudre des demandes individuelles.

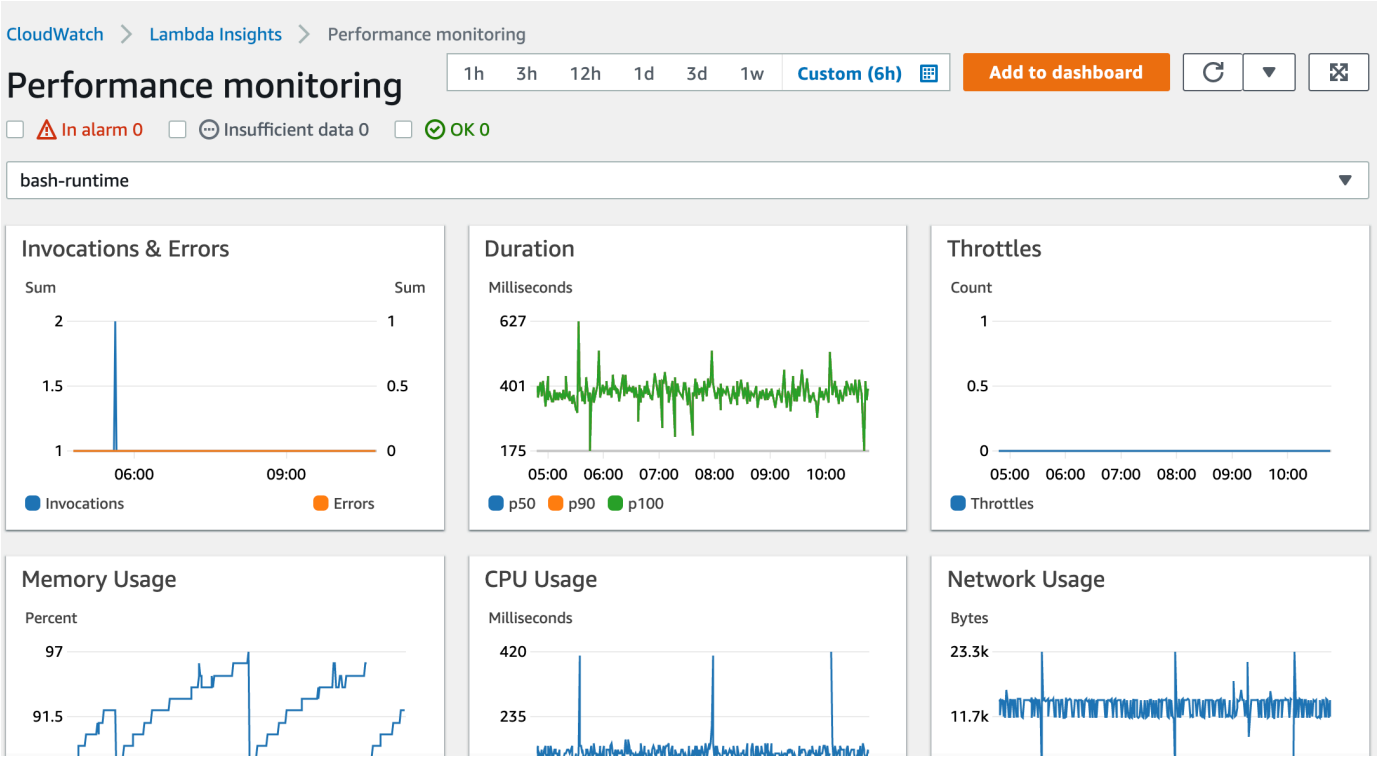
Pour afficher les métriques d'environnement d'exécution pour toutes les fonctions

1. Ouvrez la page Multi-function (Multifonction) dans la console CloudWatch.
2. Choisissez parmi les plages de temps prédéfinies ou choisissez une plage de temps personnalisée.
3. (Facultatif) Choisissez Add to dashboard (Ajouter au tableau de bord) pour ajouter les widgets à votre tableau de bord CloudWatch.



Pour afficher les métriques d'environnement d'exécution d'une fonction unique

1. Ouvrez la page Single-function (Fonction unique) dans la console CloudWatch.
2. Choisissez parmi les plages de temps prédéfinies ou choisissez une plage de temps personnalisée.
3. (Facultatif) Choisissez Add to dashboard (Ajouter au tableau de bord) pour ajouter les widgets à votre tableau de bord CloudWatch.



Pour amples d'informations, consultez Création et utilisation de widgets sur des tableaux de bord CloudWatch.

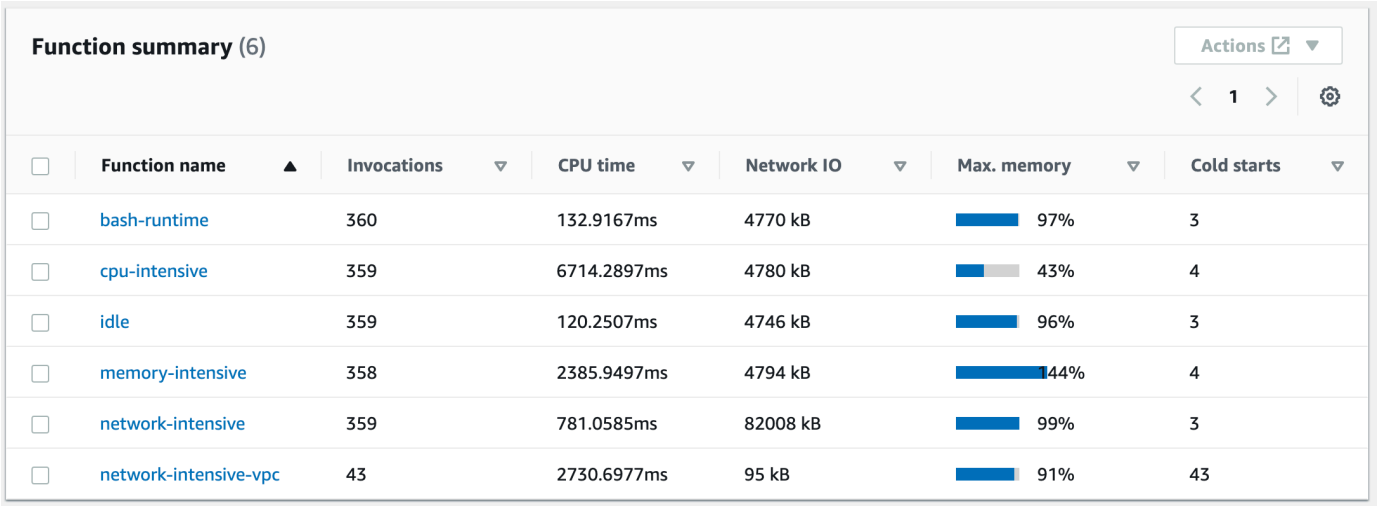
Exemple de flux de travail permettant de détecter les anomalies de fonction

Vous pouvez utiliser la vue d'ensemble multifonction sur le tableau de bord Lambda Insights pour identifier et détecter des anomalies de mémoire de calcul liées à votre fonction. Par exemple, si la vue d'ensemble multifonction indique qu'une fonction utilise une grande quantité de mémoire, vous pouvez afficher les métriques détaillées d'utilisation de la mémoire dans le volet Memory Usage (Utilisation de la mémoire). Vous pouvez ensuite accéder au tableau de bord des métriques pour activer la détection d'anomalies ou créer une alarme.

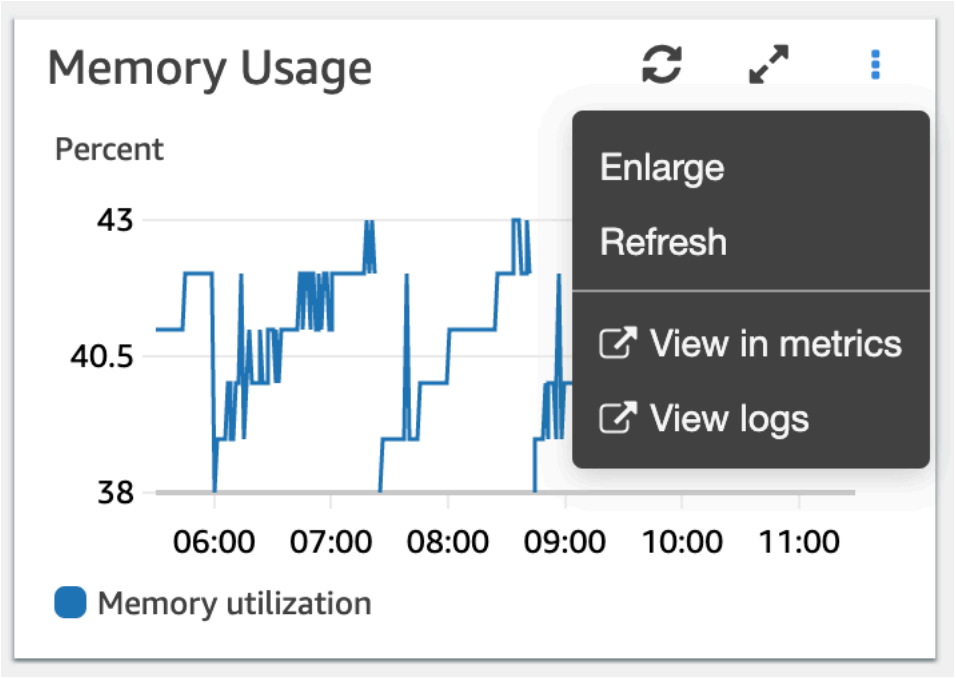
Pour activer la détection d'anomalies pour une fonction

1. Ouvrez la page Multi-function (Multifonction) dans la console CloudWatch.
2. Sous Function summary (Récapitulatif de fonction), choisissez le nom de votre fonction.

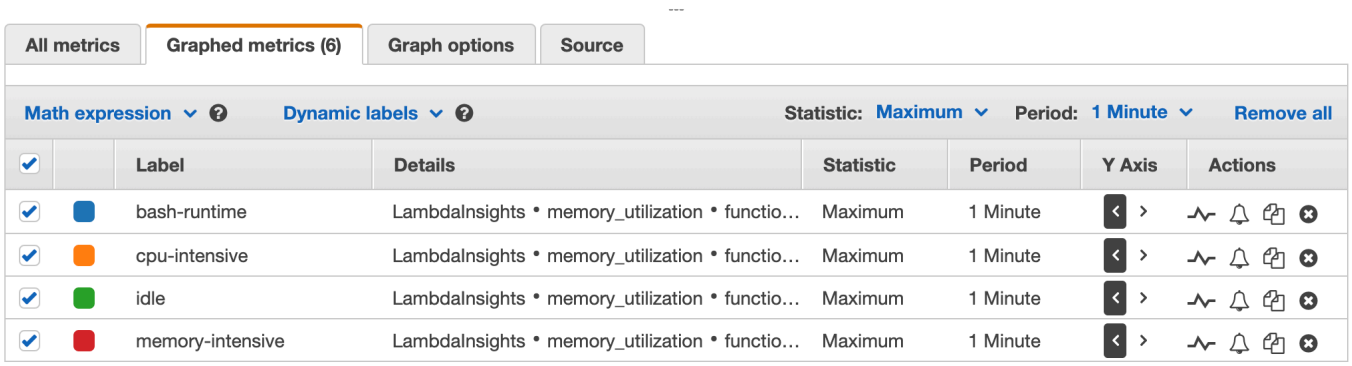
   La vue de fonction unique s'ouvre avec les métriques d'environnement d'exécution de la fonction.



3. Dans le volet Memory Usage (Utilisation de la mémoire) sélectionnez les trois points verticaux, puis choisissez View in metrics (Afficher dans les métriques) pour ouvrir le tableau de bord Metrics (Métriques).



4. Dans l'onglet Graphique des métriques, dans la colonne Actions, choisissez la première icône pour activer la détection d'anomalies pour la fonction.



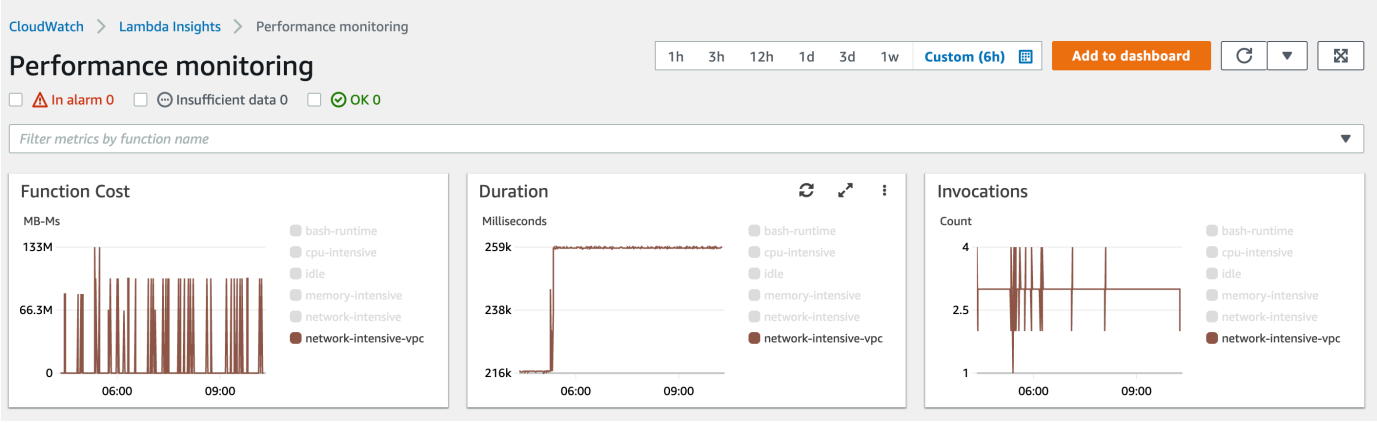
Pour plus d'informations, consultez Utilisation de la détection d'anomalies CloudWatch.

Exemple de flux de travail utilisant des demandes pour dépanner une fonction

Vous pouvez utiliser la vue de fonction unique sur le tableau de bord Lambda Insights pour identifier la cause première d'un pic de durée de fonction. Par exemple, si la vue d'ensemble multifonction indique une augmentation importante de la durée de la fonction, vous pouvez suspendre ou choisir chaque fonction dans le volet Durée afin de déterminer la fonction à l'origine de l'augmentation. Vous pouvez ensuite accéder à la vue de fonction unique et consulter les journaux d'application pour déterminer la cause principale.

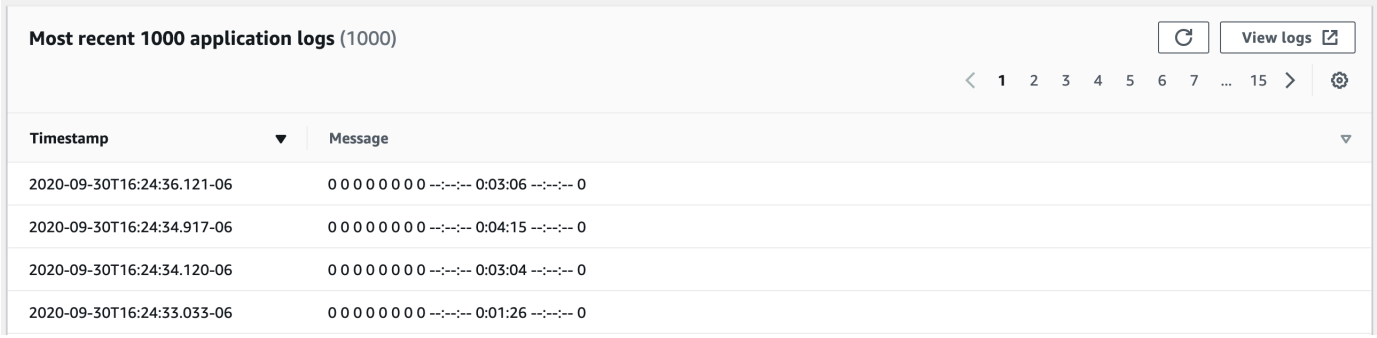
Pour exécuter des requêtes sur une fonction

1. Ouvrez la page Multi-function (Multifonction) dans la console CloudWatch.
2. Dans le volet Durée choisissez votre fonction pour filtrer les métriques de durée.

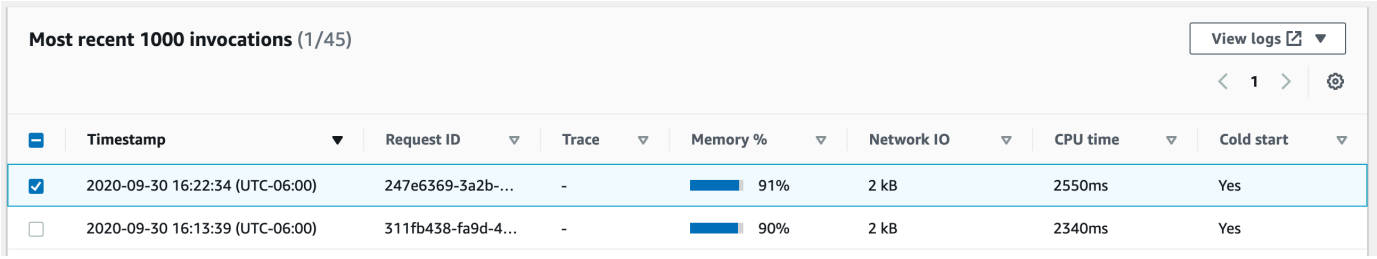


3. Ouvrez la page Fonction unique.

4. Choisissez la liste déroulante Filter metrics by function name (Filtrer les métriques par nom de fonction), puis choisissez votre fonction.
5. Pour afficher les 1000 journaux d'application les plus récents, choisissez l'onglet Application logs (Journaux d'application).
6. Examinez l'horodatage et le message pour identifier la demande d'invocation que vous souhaitez résoudre.

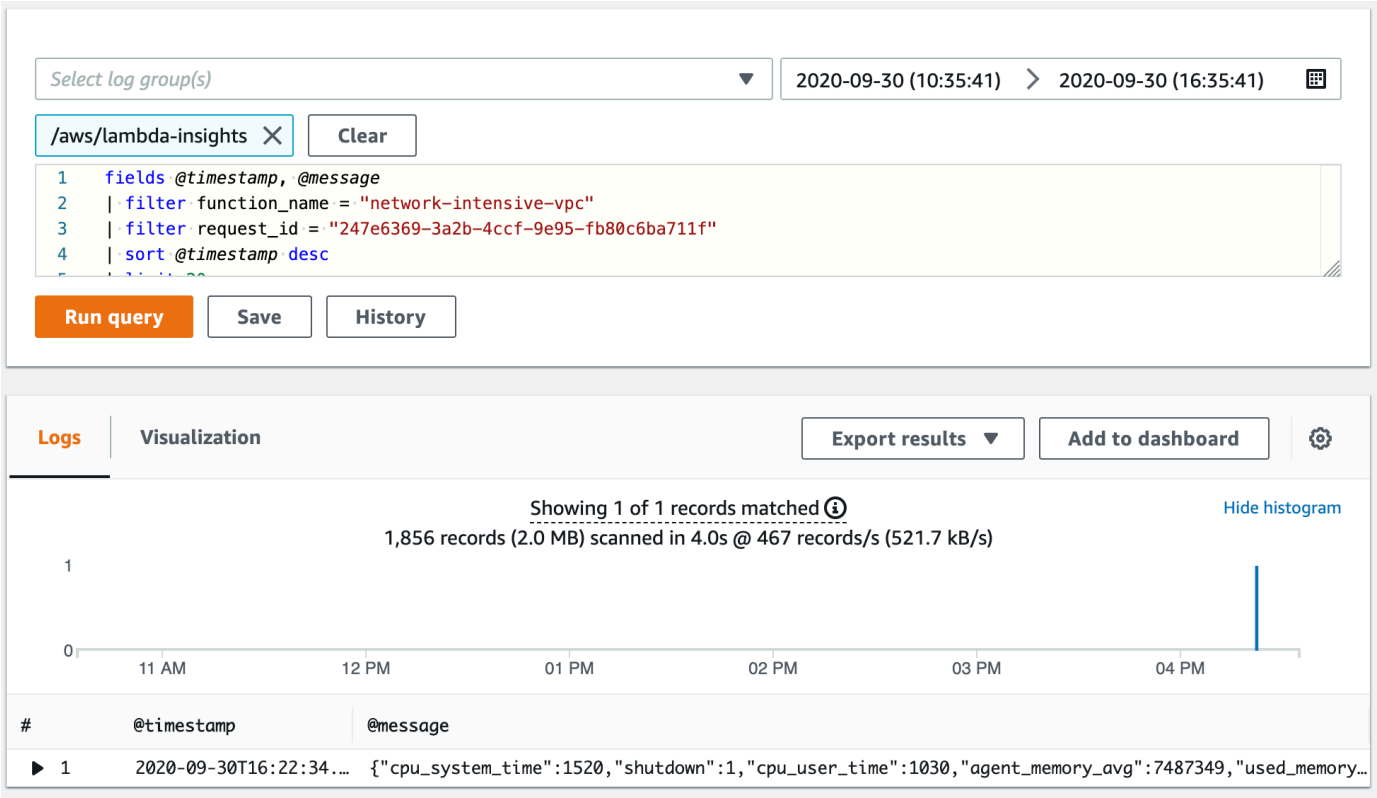


7. Pour afficher les Most recent 1000 invocations (1000 appels les plus récents), choisissez l'onglet Appels.
8. Sélectionnez l'horodatage ou le message pour la demande d'invocation que vous souhaitez résoudre.



9. Choisissez la liste déroulante Afficher les journaux, puis Afficher les journaux de performance.

   Une requête générée automatiquement pour votre fonction s'ouvre dans le tableau de bord Logs Insights.
10. Choisissez Exécuter la requête pour générer un message Journaux pour la demande d'invocation.



Quelle est la prochaine étape ?

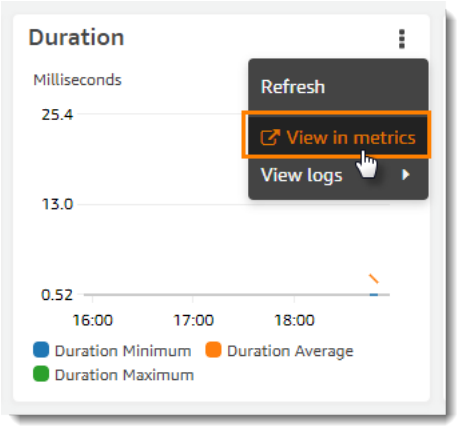
- Découvrez comment créer un tableau de bord CloudWatch Logs dans la section Créer un tableau de bord du Guide de l'utilisateur Amazon CloudWatch.
- Découvrez comment ajouter des requêtes à un tableau de bord CloudWatch Logs dans la section Ajouter une requête au tableau de bord ou exporter les résultats de requête du Manuel du développeur Amazon CloudWatch.

Surveillance des applications Lambda

La section Applications de la console Lambda inclut un onglet Surveillance dans lequel vous pouvez consulter un tableau de bord Amazon CloudWatch avec des métriques agrégées pour les ressources de votre application.

Pour surveiller une application Lambda

1. Ouvrez la page Applications de la console Lambda.
2. Choisissez Surveillance.
3. Pour afficher plus de détails sur les métriques dans un graphique, choisissez Afficher dans les métriques dans le menu déroulant.



   Le graphique s'affiche dans un nouvel onglet, avec les métriques correspondantes répertoriées sous le graphique. Vous pouvez personnaliser votre affichage de ce graphique, en modifiant les métriques et les ressources affichées, les statistiques, la période, et d'autres facteurs, afin de mieux comprendre la situation actuelle.

Par défaut, la console Lambda affiche un tableau de bord de base. Vous pouvez personnaliser cette page en ajoutant un ou plusieurs tableaux de bord Amazon CloudWatch à votre modèle d'application avec le type de ressource AWS::CloudWatch::Dashboard. Lorsque votre modèle inclut un ou plusieurs tableaux de bord, la page affiche vos tableaux de bord à la place du tableau de bord par défaut. Vous pouvez basculer entre les tableaux de bord avec le menu déroulant en haut à droite de la page. L'exemple suivant permet de créer un tableau de bord avec un seul widget qui représente graphiquement le nombre d'appels d'une fonction nommée `my-function`.

Example modèle de tableau de bord de fonction

```
Resources:
  MyDashboard:
    Type: AWS::CloudWatch::Dashboard
    Properties:
      DashboardName: my-dashboard
      DashboardBody: |
        {
            "widgets": [
                {
                    "type": "metric",
                    "width": 12,
                    "height": 6,
                    "properties": {
                        "metrics": [
                            [
                                "AWS/Lambda",
                                "Invocations",
                                "FunctionName",
                                "my-function",
                                {
                                    "stat": "Sum",
                                    "label": "MyFunction"
                                }
                            ],
                            [
                                {
                                    "expression": "SUM(METRICS())",
                                    "label": "Total Invocations"
                                }
                            ]
                        ],
                        "region": "us-east-1",
                        "title": "Invocations",
                        "view": "timeSeries",
                        "stacked": false
                    }
                }
            ]
        }
```

Pour plus d'informations sur la création des widgets et des tableaux de bord CloudWatch, consultez Syntaxe et structure du corps d'un tableau de bord dans la Référence d'API Amazon CloudWatch.

Surveillance des performances d'applications avec la vigie applicative Amazon CloudWatch

La vigie applicative Amazon CloudWatch est une solution de surveillance des performances des applications (APM) qui permet aux développeurs et aux opérateurs de surveiller l'état et les performances de leurs applications sans serveur créées à l'aide de Lambda. Vous pouvez activer la vigie applicative en un clic depuis la console Lambda, et vous n'avez pas besoin d'ajouter de code d'instrumentation ou de dépendances externes à votre fonction Lambda. Une fois que vous avez activé la vigie applicative, vous pouvez afficher toutes les métriques et suivis collectés dans la console CloudWatch. Cette page explique comment activer et afficher les données de télémétrie de la vigie applicative pour vos applications.

Rubriques

- Mode d'intégration de la vigie applicative à Lambda
- Tarification
- Environnements d'exécution pris en charge
- Activation de la vigie applicative dans la console Lambda
- Utilisation du tableau de bord de la vigie applicative

Mode d'intégration de la vigie applicative à Lambda

La vigie applicative instrumente automatiquement vos fonctions Lambda à l'aide des bibliothèques AWS Distro for OpenTelemetry (ADOT) améliorées, fournies via une couche Lambda. La vigie applicative lit les données collectées par la couche et génère des tableaux de bord contenant des indicateurs de performance clés pour vos applications.

Vous pouvez attacher cette couche en un clic en activant la vigie applicative dans la console Lambda. Lorsque vous activez la vigie applicative à partir de la console, Lambda effectue les opérations suivantes en votre nom :

- Mise à jour du rôle d'exécution de la fonction pour inclure `CloudWatchLambdaApplicationSignalsExecutionRolePolicy`. Cette politique fournit un accès en écriture à AWS X-Ray et aux groupes de journaux CloudWatch utilisés pour la vigie applicative.
- Ajout d'une couche à votre fonction qui instrument automatiquement la fonction pour capturer les données de télémétrie telles que les requêtes, la disponibilité, la latence, les erreurs et les défauts.

Pour garantir le bon fonctionnement de la vigie applicative, supprimez tout code d'instrumentation du kit SDK X-Ray existant de votre fonction. Le code d'instrumentation personnalisé du kit SDK X-Ray peut interférer avec l'instrumentation fournie par la couche.

- Ajout de la variable d'environnement `AWS_LAMBDA_EXEC_WRAPPER` à votre fonction et définition de sa valeur sur `/opt/otel-instrument`. Cette variable d'environnement modifie le comportement de démarrage de votre fonction afin d'utiliser la couche de la vigie applicative. Elle est requise pour une instrumentation appropriée. Si cette variable d'environnement existe déjà, assurez-vous qu'elle est définie sur la valeur adéquate.

Tarification

L'utilisation de la vigie applicative pour vos fonctions Lambda entraîne des coûts. Pour de plus amples informations, consultez la Tarification d'Amazon CloudWatch.

Environnements d'exécution pris en charge

L'intégration de la vigie applicative à Lambda fonctionne avec les environnements d'exécution suivants :

- .NET 8
- Java 11
- Java 17
- Java 21
- Python 3.10
- Python 3.11
- Python 3.12
- Python 3.13
- Node.js 18.x
- Node.js 20.x
- Node.js 22.x

Activation de la vigie applicative dans la console Lambda

Vous pouvez activer la vigie applicative sur n'importe quelle fonction Lambda existante à l'aide d'un environnement d'exécution compatible. Les étapes suivantes expliquent comment activer la vigie applicative en un clic dans la console Lambda.

Pour activer la vigie applicative dans la console Lambda

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez votre fonction.
3. Choisissez l'onglet Configuration.
4. Dans le menu de gauche, sélectionnez Outils de surveillance et d'exploitation.
5. Dans le volet Outils de surveillance supplémentaires, choisissez Modifier.
6. Sous Vigie applicative CloudWatch et AWS X-Ray, et sous Vigie applicative, choisissez Activer.
7. Choisissez Enregistrer.

Si c'est la première fois que vous activez la vigie applicative pour votre fonction, vous devez également effectuer une configuration unique de découverte de service pour la vigie applicative dans la console CloudWatch. Une fois que vous avez terminé cette configuration unique de découverte de service, la vigie applicative découvre automatiquement toutes les fonctions Lambda supplémentaires pour lesquelles vous activez la vigie applicative, dans toutes les régions.

> **Note**
>
> Après avoir invoqué votre fonction mise à jour, il peut s'écouler jusqu'à dix minutes avant que les données de service n'apparaissent dans le tableau de bord de la vigie applicative de la console CloudWatch.

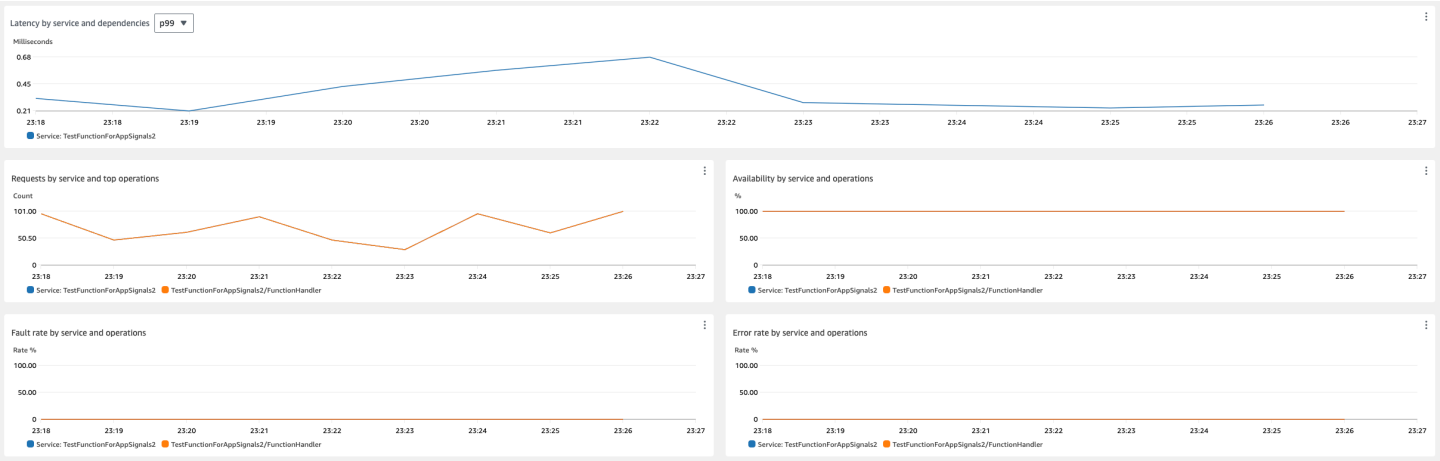
Utilisation du tableau de bord de la vigie applicative

Après avoir activé la vigie applicative pour votre fonction, vous pouvez visualiser les métriques de votre application dans la console CloudWatch. Vous pouvez rapidement consulter le tableau de bord de la vigie applicative associé depuis la console Lambda en procédant comme suit :

Pour consulter le tableau de bord de la vigie applicative de votre fonction

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez votre fonction.
3. Choisissez l'onglet Monitor (Surveiller).
4. Cliquez sur le bouton Afficher la vigie applicative. Vous accédez ainsi directement à la vue d'ensemble de la vigie applicative pour votre service dans la console CloudWatch.

Par exemple, la capture d'écran suivante montre les métriques relatives à la latence, au nombre de requêtes, à la disponibilité, au taux de défaillance et au taux d'erreur pour une fonction sur une période de dix minutes.



Pour tirer le meilleur parti de votre intégration à la vigie applicative, vous pouvez créer des objectifs de niveau de service (SLO) pour votre application. Par exemple, vous pouvez créer des SLO de latence pour garantir que votre application répond rapidement aux requêtes des utilisateurs, et des SLO de disponibilité pour suivre la durée de fonctionnement. Les SLO peuvent vous aider à détecter la dégradation des performances ou les pannes avant qu'elles n'affectent vos utilisateurs. Pour plus d'informations, consultez Service level objectives (SLOs) dans le Guide de l'utilisateur Amazon CloudWatch.

Déboguer à distance des fonctions Lambda avec Visual Studio Code

Grâce à la fonction de débogage à distance intégrée à AWS Toolkit for Visual Studio Code, vous pouvez déboguer vos fonctions Lambda exécutées directement dans le cloud AWS. Cela est utile lorsque vous étudiez des problèmes difficiles à reproduire localement ou à diagnostiquer uniquement à l'aide de journaux.

Grâce au débogage à distance, vous pouvez :

- Définir des points d'arrêt dans le code de votre fonction Lambda.
- Exécuter le code étape par étape en temps réel.
- Inspecter les variables et leur état pendant l'exécution.
- Déboguer les fonctions Lambda déployées sur AWS, y compris celles qui se trouvent dans des VPC ou avec des autorisations IAM spécifiques.

Environnements d'exécution pris en charge

Le débogage à distance est pris en charge pour les environnements d'exécution suivants :

- Python (AL2023)
- Java
- JavaScript/Node.js (AL2023)

> **Note**
>
> Le débogage à distance est pris en charge pour les architectures x86_64 et arm64.

Sécurité et débogage à distance

Le débogage à distance fonctionne dans les limites de sécurité Lambda existantes.
Les utilisateurs peuvent associer des couches à une fonction à l'aide de l'autorisation `UpdateFunctionConfiguration`, qui permet déjà d'accéder aux variables d'environnement et à la configuration de la fonction. Le débogage à distance ne s'étend pas au-delà de ces autorisations

existantes. Il ajoute plutôt des contrôles de sécurité supplémentaires grâce à un tunneling sécurisé et à une gestion automatique des sessions. En outre, le débogage à distance est une fonctionnalité entièrement contrôlée par le client nécessitant des autorisations et des actions explicites :

- Création d'un tunnel sécurisé IoT : AWS Toolkit doit créer un tunnel sécurisé IoT, ce qui n'est possible qu'avec l'autorisation explicite de l'utilisateur en utilisant `iot:OpenTunnel`.
- Attachement de la couche de débogage et gestion des jetons : le processus de débogage assure la sécurité via les contrôles suivants :
  - La couche de débogage doit être attachée à la fonction Lambda, et ce processus nécessite les autorisations suivantes : `lambda:UpdateFunctionConfiguration` et `lambda:GetLayerVersion`.
  - Un jeton de sécurité (généré via `iot:OpenTunnel`) doit être mis à jour dans la variable d'environnement de la fonction avant chaque session de débogage, ce qui nécessite également `lambda:UpdateFunctionConfiguration`.
  - Pour des raisons de sécurité, ce jeton est automatiquement pivoté, et la couche de débogage est automatiquement supprimée à la fin de chaque session de débogage et ne peut pas être réutilisée.

> **Note**
>
> Le débogage à distance est pris en charge pour les architectures x86_64 et arm64.

Prérequis

Avant de commencer le débogage à distance, vérifiez que vous disposez des éléments suivants :

1. Une fonction Lambda déployée sur votre compte AWS.
2. AWS Toolkit for Visual Studio Code. Consultez Configuration d'AWS Toolkit for Visual Studio Code pour les instructions d'installation.
3. La version d'AWS Toolkit que vous avez installée est la version 3.69.0 ou une version ultérieure.
4. Informations d'identification AWS configurées dans AWS Toolkit for Visual Studio Code. Pour de plus amples informations, consultez Authentification et contrôle d'accès.

Débogage à distance de fonctions Lambda

Pour démarrer une session de débogage à distance, procédez comme suit :

1. Ouvrez l'explorateur AWS dans VS Code en sélectionnant l'icône AWS dans la barre latérale gauche.
2. Développez la section Lambda pour voir vos fonctions.
3. Cliquez sur la fonction que vous souhaitez déboguer avec le bouton droit de la souris.
4. Dans le menu contextuel, sélectionnez Invoquer à distance.
5. Dans la fenêtre d'invocation qui s'ouvre, cochez la case Activer le débogage.
6. Cliquez sur Invoquer pour démarrer la session de débogage à distance.

> **Note**
>
> Les fonctions Lambda ont une limite combinée de 250 Mo pour le code de fonction et toutes les couches associées. La couche de débogage à distance ajoute environ 40 Mo à la taille de votre fonction.

Une session de débogage à distance se termine quand vous :

- Choisissez Supprimer la configuration de débogage sur l'écran de configuration de l'invocation à distance.
- Sélectionnez l'icône de déconnexion dans les commandes de débogage de VS Code.
- Sélectionnez le fichier de gestionnaire dans l'éditeur VS Code.

> **Note**
>
> La couche de débogage est automatiquement supprimée après 60 secondes d'inactivité après votre dernière invocation.

Désactiver le débogage à distance

Vous pouvez désactiver cette fonctionnalité de trois manières :

- Refuser les mises à jour de fonction : définissez `lambda:UpdateFunctionConfiguration` sur `deny`.
- Restreindre les autorisations IoT : refusez les autorisations IoT
- Bloquer les couches de débogage : refusez `lambda:GetLayerVersion` pour les ARN suivants :
  - `arn:aws:lambda:*:*:layer:LDKLayerX86:*`
  - `arn:aws:lambda:*:*:layer:LDKLayerArm64:*`

> **Note**
>
> La désactivation de cette fonctionnalité empêche l'ajout de la couche de débogage lors des mises à jour de configuration des fonctions.

Informations supplémentaires

Pour plus d'informations sur l'utilisation de Lambda dans VS Code, reportez-vous à la section Développement de fonctions Lambda localement avec VS Code.

Pour obtenir des instructions détaillées sur le dépannage, les cas d'utilisation avancés et la disponibilité régionale, consultez Débogage à distance des fonctions Lambda dans le Guide de l'utilisateur AWS Toolkit for Visual Studio Code.

Gestion des dépendances Lambda à l’aide de couches

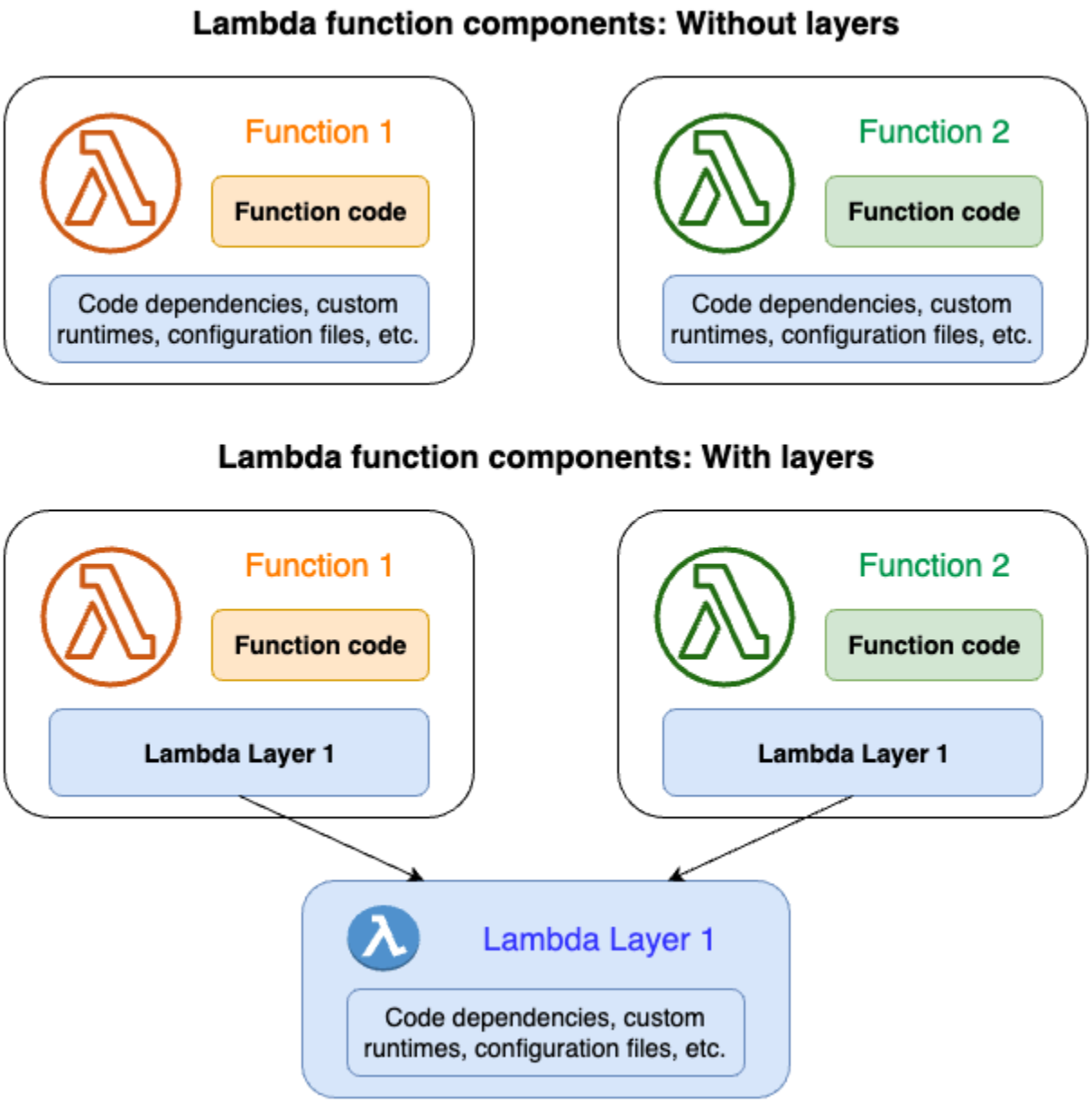
Une couche Lambda est une archive de fichier .zip qui contient du code ou des données supplémentaires. Les couches contiennent généralement des dépendances de bibliothèque, une exécution personnalisée, ou des fichiers de configuration.

Vous pouvez envisager d'utiliser des couches pour plusieurs raisons :

- Pour réduire la taille de vos packages de déploiement. Au lieu d'inclure toutes vos dépendances de fonctions avec votre code de fonction dans votre package de déploiement, placez-les dans une couche. Cela permet de maintenir la taille et l'organisation des packages de déploiement.
- Pour séparer la logique des fonctions de base des dépendances. Avec les couches, vous pouvez mettre à jour les dépendances de vos fonctions indépendamment du code de votre fonction, et vice versa. Cela favorise la séparation des préoccupations et vous aide à vous concentrer sur la logique de votre fonction.
- Pour partager les dépendances entre plusieurs fonctions. Après avoir créé une couche, vous pouvez l'appliquer à un certain nombre de fonctions de votre compte. Sans couches, vous devez inclure les mêmes dépendances dans chaque package de déploiement individuel.
- Pour utiliser l'éditeur de code de la console Lambda. L'éditeur de code est un outil utile pour tester rapidement les mises à jour mineures du code des fonctions. Toutefois, vous ne pouvez pas utiliser l'éditeur si la taille de votre package de déploiement est trop importante. L'utilisation de couches réduit la taille de votre package et peut débloquer l'utilisation de l'éditeur de code.
- Pour verrouiller une version de SDK intégrée.Les kits SDK intégrés peuvent être modifiés sans préavis au fur et à mesure qu’AWS publie de nouveaux services et fonctionnalités. Vous pouvez verrouiller une version du kit SDK en créant une couche Lambda avec la version spécifique requise. La fonction utilise alors toujours la version de la couche, même si la version intégrée au service change.

Si vous utilisez des fonctions Lambda dans Go ou Rust, nous vous déconseillons d’utiliser des couches. Le code des fonctions Go et Rust est fourni sous la forme d’un exécutable, qui contient le code compilé de la fonction ainsi que toutes ses dépendances. Placer vos dépendances dans une couche oblige votre fonction à charger manuellement des assemblages supplémentaires pendant la phase d’initialisation, ce qui peut augmenter les temps de démarrage à froid. Pour des performances optimales pour les fonctions Go et Rust, incluez vos dépendances dans votre package de déploiement.

Le diagramme suivant illustre les différences architecturales de haut niveau entre deux fonctions qui partagent des dépendances. L'un utilise des couches Lambda, l'autre non.



Lorsque vous ajoutez une couche à une fonction, Lambda extrait le contenu de la couche dans le répertoire `/opt` de l'environnement d'exécution de votre fonction. Toutes les exécutions Lambda prises en charge en mode natif incluent des chemins vers des répertoires spécifiques dans le répertoire `/opt`. Cela permet à votre fonction d'accéder au contenu de votre couche. Pour plus d'informations sur ces chemins spécifiques et sur la manière d'empaqueter correctement vos couches, consultez the section called “Empaquetage des couches”.

Vous pouvez inclure jusqu'à cinq couches par fonction. En outre, vous ne pouvez utiliser des couches qu'avec des fonctions Lambda déployées en tant qu'archive de fichiers .zip. Pour des fonctions définies en tant qu'image de conteneur, créez un package avec votre exécution préférée et toutes les dépendances de code lorsque vous créez l'image de conteneur. Pour plus d'informations, consultez Utilisation de couches et d'extensions Lambda dans des images conteneurs sur le blog AWS Compute.

Rubriques

- Utilisation des couches
- Couches et versions de couches
- Empaquetage du contenu de votre couche
- Création et suppression de couches dans Lambda
- Ajout de couches aux fonctions
- Utilisation de AWS CloudFormation avec des couches
- Utilisation de AWS SAM avec des couches

Utilisation des couches

Pour créer une couche, empaquetez vos dépendances dans un fichier .zip, de la même manière que vous créer un package de déploiement normal. Plus précisément, le processus général de création et d'utilisation de couches comporte les trois étapes suivantes :

- Empaquetez d'abord le contenu de votre couche. Cela implique de créer une archive de fichiers .zip. Pour de plus amples informations, consultez the section called “Empaquetage des couches”.
- Créez ensuite la couche dans Lambda. Pour de plus amples informations, consultez the section called “Création et suppression de couches”.
- Ajoutez la couche à vos fonctions). Pour de plus amples informations, consultez the section called “Ajout de couches”.

Couches et versions de couches

Une version de couche est un instantané immuable d'une version spécifique d'une couche. Lorsque vous créez une nouvelle couche, Lambda crée une nouvelle version de couche avec un numéro de

version de 1. Chaque fois que vous publiez une mise à jour de la couche, Lambda incrémente le numéro de version et crée une nouvelle version de couche.

Chaque version de couche est identifiée par un Amazon Resource Name (ARN) unique. Lorsque vous ajoutez une couche à la fonction, vous devez spécifier la version de couche exacte que vous voulez utiliser (par exemple `arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`).

Empaquetage du contenu de votre couche

Une couche Lambda est une archive de fichier .zip qui contient du code ou des données supplémentaires. Les couches contiennent généralement des dépendances de bibliothèque, une exécution personnalisée, ou des fichiers de configuration.

Cette section explique comment empaqueter correctement le contenu de votre couche. Pour plus d'informations conceptuelles sur les couches et les raisons pour lesquelles vous pourriez envisager de les utiliser, consultez Couches Lambda.

La première étape de la création d'une couche consiste à regrouper l'ensemble du contenu de la couche dans une archive de fichiers .zip. Parce que les fonctions Lambda s'exécutent sur Amazon Linux, le contenu de votre couche doit pouvoir être compilé et construit dans un environnement Linux.

Pour vous assurer que le contenu de votre couche fonctionne correctement dans un environnement Linux, nous vous recommandons de créer le contenu de votre couche à l'aide d'un outil comme Docker.

Rubriques

- Chemins d'accès de couche pour chaque exécution Lambda

Chemins d'accès de couche pour chaque exécution Lambda

Lorsque vous ajoutez une couche à une fonction, Lambda charge le contenu de la couche dans le répertoire `/opt` de cet environnement d'exécution. Pour chaque exécution Lambda, la variable `PATH` inclut déjà des chemins de dossiers spécifiques dans le répertoire `/opt`. Pour garantir que Lambda capture le contenu de votre couche, le fichier .zip de votre couche doit avoir ses dépendances un des chemins de dossier suivants :

Runtime	Chemin
Node.js	`nodejs/node_modules`
	`nodejs/node18/node_modules` (`NODE_PATH`)
	`nodejs/node20/node_modules` (`NODE_PATH`)
	`nodejs/node22/node_modules` (`NODE_PATH`)

Runtime	Chemin
Python	`python`
	`python/lib/` *`python3.x`* `/site-packages` (répertoires de site)
Java	`java/lib` (`CLASSPATH`)
Ruby	`ruby/gems/3.4.0` (`GEM_PATH`)
	`ruby/lib` (`RUBYLIB`)
Toutes les exécutions	`bin` (`PATH`)
	`lib` (`LD_LIBRARY_PATH`)

Les exemples suivants montrent comment structurer les dossiers dans votre couche d'archive .zip.

Node.js

Example structure de fichiers du kit SDK AWS X-Ray pour Node.js

```
xray-sdk.zip
# nodejs/node_modules/aws-xray-sdk
```

Python

Example

```
python/              # Required top-level directory
### requests/
### boto3/
### numpy/
### (dependencies of the other packages)
```

Ruby

Example structure de fichiers pour la gem JSON

```
json.zip
```

```
# ruby/gems/3.4.0/
                | build_info
                | cache
                | doc
                | extensions
                | gems
                | # json-2.1.0
                # specifications
                  # json-2.1.0.gemspec
```

Java

Example structure de fichiers pour le fichier JAR Jackson

```
layer_content.zip
# java
    # lib
        # jackson-core-2.17.0.jar
        # <other potential dependencies>
        # ...
```

All

Example structure de fichiers pour la bibliothèque jq

```
jq.zip
# bin/jq
```

Pour obtenir des instructions spécifiques au langage et relatives à l'empaquetage, à la création et à l'ajout d'une couche, reportez-vous aux pages suivantes :

- Node.js – the section called "Layers"
- Python – the section called "Layers"
- Ruby – the section called "Couches"
- Java : the section called "Layers"

Nous déconseillons d'utiliser des couches pour gérer les dépendances des fonctions Lambda écrites en Go et Rust. Cela est dû au fait que les fonctions Lambda écrites dans ces langages sont compilées en un seul exécutable, que vous fournissez à Lambda lorsque vous déployez votre

fonction. Cet exécutable contient votre code de fonction compilé, ainsi que toutes ses dépendances. L'utilisation de couches complique non seulement ce processus, mais entraîne également une augmentation des temps de démarrage à froid, car vos fonctions doivent charger manuellement des assemblages supplémentaires en mémoire pendant la phase d'initialisation.

Pour utiliser des dépendances externes avec vos fonctions Lambda Go et Rust, incluez-les directement dans votre package de déploiement.

Création et suppression de couches dans Lambda

Une couche Lambda est une archive de fichier .zip qui contient du code ou des données supplémentaires. Les couches contiennent généralement des dépendances de bibliothèque, une exécution personnalisée, ou des fichiers de configuration.

Cette section explique comment créer et supprimer des couches dans Lambda. Pour plus d'informations conceptuelles sur les couches et les raisons pour lesquelles vous pourriez envisager de les utiliser, consultez Couches Lambda.

Une fois que vous avez empaqueté le contenu de votre couche, l'étape suivante consiste à créer la couche dans Lambda. Cette section explique comment créer et supprimer des couches à l'aide de la console Lambda ou de l'API Lambda uniquement. Pour créer une couche à l'aide d' AWS CloudFormation, consultez the section called "Couches avec CloudFormation". Pour créer une couche à l'aide d' AWS Serverless Application Model (AWS SAM), consultez the section called "Couches avec AWS SAM".

Rubriques

- Création d'une couche
- Suppression d'une version de couche

Création d'une couche

Pour créer une couche, vous pouvez soit charger l'archive du fichier .zip depuis votre ordinateur local, soit depuis Amazon Simple Storage Service (Amazon S3). Lors de la configuration de l'environnement d'exécution pour la fonction, Lambda extrait le contenu de la couche dans le répertoire `/opt`.

Les couches peuvent avoir une ou plusieurs versions de couche. Lorsque vous créez une couche, Lambda définit la version de la couche sur version 1. Vous pouvez modifier les autorisations sur une version de couche existante à tout moment. Toutefois, pour mettre à jour le code ou apporter d'autres modifications de configuration, vous devez créer une nouvelle version de la couche.

Pour créer une couche (console)

1. Ouvrez la page Couches de la console Lambda.
2. Sélectionnez Créer un calque.
3. Sous Configuration de la couche, dans Nom, nommez votre couche.

4. (Facultatif) Dans le champ Description, saisissez une description pour le calque.
5. Pour télécharger le code de votre couche, effectuez l'une des opérations suivantes :
   - Pour charger un fichier .zip à partir de votre ordinateur, choisissez Charger un fichier .zip. Puis, sélectionnez Charger pour sélectionner votre fichier .zip local.
   - Pour charger un fichier à partir d'Amazon S3, choisissez Charger un fichier à partir d'Amazon S3. Ensuite, pour l'URL du lien Amazon S3, saisissez un lien vers le fichier.
6. (Facultatif) Pour Architectures compatibles, choisissez une valeur ou les deux valeurs. Pour de plus amples informations, veuillez consulter the section called "Jeux d'instructions (ARM/x86)".
7. (Facultatif) Pour Exécutions compatibles, choisissez les exécutions avec lesquelles votre couche est compatible.
8. (Facultatif) Pour Licence, saisissez toutes les informations de licence nécessaires.
9. Choisissez Créer.

Vous pouvez également exécuter la commande publish-layer-version AWS Command Line Interface (CLI). Exemple :

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip --
compatible-runtimes python3.14
```

Chaque fois que vous exécutez `publish-layer-version`, Lambda crée une nouvelle version de la couche.

Suppression d'une version de couche

Pour supprimer une version de couche, utilisez l'opération DeleteLayerVersionAPI. Par exemple, exécutez la delete-layer-version AWS CLI commande avec le nom de couche et la version de couche spécifiés.

```
aws lambda delete-layer-version --layer-name my-layer --version-number 1
```

Lorsque vous supprimez une version de couche, vous ne pouvez plus configurer de fonction Lambda pour l'utiliser. En revanche, toute fonction qui utilise déjà la version continue d'y avoir accès. De plus, Lambda ne réutilise jamais les numéros de version pour le nom d'une couche.

Lors du calcul des quotas, la suppression d'une version de couche signifie qu'elle n'est plus prise en compte dans le quota par défaut de 75 Go pour le stockage des fonctions et des couches. Toutefois,

pour les fonctions qui utilisent une version de couche supprimée, le contenu de la couche est toujours pris en compte dans le quota de taille du package de déploiement de la fonction (c'est-à-dire 250 Mo pour les archives de fichiers .zip).

Ajout de couches aux fonctions

Une couche Lambda est une archive de fichier .zip qui contient du code ou des données supplémentaires. Les couches contiennent généralement des dépendances de bibliothèque, une exécution personnalisée, ou des fichiers de configuration.

Cette section explique comment ajouter une couche à une fonction Lambda. Pour plus d'informations conceptuelles sur les couches et les raisons pour lesquelles vous pourriez envisager de les utiliser, consultez Couches Lambda.

Avant de pouvoir configurer une fonction Lambda pour utiliser une couche, vous devez :

- Empaqueter le contenu de votre couche
- Créer une couche dans Lambda
- Assurez-vous que vous êtes autorisé à appeler l'GetLayerVersionAPI sur la version de la couche. Pour les fonctions de votre Compte AWS, vous devez disposer de cette autorisation dans votre politique d'utilisation. Pour utiliser une couche dans un autre compte, le propriétaire de ce compte doit accorder l'autorisation à votre compte dans une stratégie basée sur les ressources. Pour obtenir des exemples, consultez the section called "Accès de la couche pour les autres comptes".

Vous pouvez ajouter jusqu'à cinq couches à une fonction Lambda. La taille totale décompressée de la fonction avec toutes les couches ne peut pas dépasser le quota de taille de package de déploiement décompressé de 250 Mo. Pour de plus amples informations, veuillez consulter Quotas Lambda.

Vos fonctions peuvent continuer à utiliser n'importe quelle version de couche que vous avez déjà ajoutée, même après la suppression de cette version de couche ou après la révocation de votre autorisation d'accès à la couche. Toutefois, vous ne pouvez pas créer une nouvelle fonction qui utilise une version de couche supprimée.

Ajouter une couche à une fonction

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez la fonction.
3. Faites défiler jusqu'à la section Couches, puis choisissez Ajouter une couche.
4. Sous Choisir une couche, choisissez une source de couche :

    a. AWS couches : choisissez dans la liste des extensions AWS gérées.

   b. Couches personnalisées : choisissez une couche créée dans votre Compte AWS.
   c. Spécifiez un ARN : pour utiliser une couche provenant d'une autre couche Compte AWS, telle qu'une extension tierce, entrez le Amazon Resource Name (ARN).
5. Choisissez Ajouter.

L'ordre dans lequel vous ajoutez les couches correspond à l'ordre dans lequel Lambda fusionne ultérieurement le contenu de la couche dans l'environnement d'exécution. Vous pouvez modifier l'ordre de fusion des couches à l'aide de la console.

Pour mettre à jour l'ordre de fusion des couches pour votre fonction (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez la fonction à configurer.
3. Sous Layers (Couches), choisissez Edit (Modifier).
4. Choisissez l'une des couches.
5. Choisissez Merge earlier (Fusionner plus tôt) ou Merge later (Fusionner plus tard) pour ajuster l'ordre des couches.
6. Choisissez Enregistrer.

Les couches sont versionnées. Le contenu de chaque version de couche est immuable. Le propriétaire d'une couche peut publier une nouvelle version de la couche pour fournir du contenu mis à jour. Vous pouvez utiliser la console pour mettre à jour la version de couche associée à vos fonctions.

Pour mettre à jour les versions des couches de votre fonction (console)

1. Ouvrez la page Couches de la console Lambda.
2. Choisissez la couche pour laquelle vous voulez mettre à jour la version.
3. Choisissez l'onglet Fonctions utilisant cette version.
4. Choisissez les fonctions que vous voulez modifier, puis cliquez sur Modifier.
5. Pour Version de la couche, choisissez la version de la couche à modifier.
6. Choisissez Update functions (Mettre à jour les fonctions).

Vous ne pouvez pas mettre à jour les versions des couches fonctionnelles d'un AWS compte à l'autre.

Recherche d’informations sur la couche

Pour trouver dans votre compte des couches compatibles avec le temps d'exécution de votre fonction, utilisez l'ListLayersAPI. Par exemple, vous pouvez utiliser la commande list-layers (AWS Command Line Interface CLI) suivante :

```
aws lambda list-layers --compatible-runtime python3.14
```

Vous devez voir des résultats similaires à ce qui suit :

```
{
    "Layers": [
        {
            "LayerName": "my-layer",
            "LayerArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-layer",
            "LatestMatchingVersion": {
                "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-
layer:2",
                "Version": 2,
                "Description": "My layer",
                "CreatedDate": "2025-04-15T00:37:46.592+0000",
                "CompatibleRuntimes": [
                    "python3.14"
                ]
            }
        }
    ]
}
```

Pour répertorier toutes les couches de votre compte, ignorez l’option `--compatible-runtime`. Les détails de la réponse indiquent la dernière version de chaque couche.

Vous pouvez également obtenir la dernière version d'une couche à l'aide de l'ListLayerVersionsAPI. Par exemple, vous pouvez utiliser la commande CLI `list-layer-versions` suivante :

```
aws lambda list-layer-versions --layer-name my-layer
```

Vous devez voir des résultats similaires à ce qui suit :

```
{
    "LayerVersions": [
        {
            "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-
layer:2",
            "Version": 2,
            "Description": "My layer",
            "CreatedDate": "2023-11-15T00:37:46.592+0000",
            "CompatibleRuntimes": [
                "java11"
            ]
        },
        {
            "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-
layer:1",
            "Version": 1,
            "Description": "My layer",
            "CreatedDate": "2023-11-15T00:27:46.592+0000",
            "CompatibleRuntimes": [
                "java11"
            ]
        }
    ]
}
```

Utilisation de AWS CloudFormation avec des couches

Vous pouvez utiliser CloudFormation pour créer une couche et l'associer à votre fonction Lambda. L'exemple de modèle suivant crée une couche nommée `my-lambda-layer` et l'attache à la fonction Lambda à l'aide de la propriété Couches.

Dans cet exemple, le modèle indique l'Amazon Resource Name (ARN) d'un rôle d'exécution IAM existant. Vous pouvez également créer un nouveau rôle d'exécution dans le modèle à l'aide de la ressource CloudFormation AWS::IAM::Role.

Votre fonction n'a pas besoin d'autorisations spéciales pour utiliser les couches.

```
---
Description: CloudFormation Template for Lambda Function with Lambda Layer
Resources:
  MyLambdaLayer:
    Type: AWS::Lambda::LayerVersion
    Properties:
      LayerName: my-lambda-layer
      Description: My Lambda Layer
      Content:
        S3Bucket: amzn-s3-demo-bucket
        S3Key: my-layer.zip
      CompatibleRuntimes:
        - python3.9
        - python3.10
        - python3.11

  MyLambdaFunction:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: my-lambda-function
      Runtime: python3.9
      Handler: index.handler
      Timeout: 10
      Role: arn:aws:iam::111122223333:role/my_lambda_role
      Layers:
        - !Ref MyLambdaLayer
```

Utilisation de AWS SAM avec des couches

Vous pouvez utiliser l'AWS Serverless Application Model (AWS SAM) pour automatiser la création de couches dans votre application. Le type de ressource `AWS::Serverless::LayerVersion` crée une version de couche que vous pouvez référencer à partir de la configuration de votre fonction Lambda.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: AWS SAM Template for Lambda Function with Lambda Layer

Resources:
  MyLambdaLayer:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: my-lambda-layer
      Description: My Lambda Layer
      ContentUri: s3://amzn-s3-demo-bucket/my-layer.zip
      CompatibleRuntimes:
        - python3.9
        - python3.10
        - python3.11

  MyLambdaFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: MyLambdaFunction
      Runtime: python3.9
      Handler: app.handler
      CodeUri: s3://amzn-s3-demo-bucket/my-function
      Layers:
        - !Ref MyLambdaLayer
```

Utilisation des extensions Lambda pour augmenter vos fonctions Lambda

Vous pouvez utiliser des extensions Lambda pour augmenter vos fonctions Lambda. Par exemple, utilisez des extensions Lambda pour intégrer des fonctions à vos outils de surveillance, d'observabilité, de sécurité et de gouvernance préférés. Vous pouvez choisir parmi un vaste éventail d'outils fournis par des partenaires AWS Lambda, ou créer vos propres extensions Lambda.

Lambda prend en charge les extensions externes et internes. Une extension externe s'exécute en tant que processus indépendant dans l'environnement d'exécution et continue à s'exécuter une fois l'invocation de fonction entièrement traité. Étant donné que les extensions s'exécutent sous forme de processus distincts, vous pouvez les écrire dans un langage différent de la fonction. Tous les Environnements d'exécution (runtimes) Lambda prennent en charge des extensions.

Une extension interne s'exécute dans le cadre du processus de runtime. Votre fonction accède aux extensions internes via des scripts encapsuleurs ou des mécanismes internes tels que `JAVA_TOOL_OPTIONS`. Pour plus d'informations, consultez Modification de l'environnement d'exécution.

Vous pouvez ajouter des extensions à une fonction à l'aide de la console Lambda, de l'AWS Command Line Interface (AWS CLI) ou de services et outils d'infrastructure en tant que code (infrastructure as code, IaC) tels que CloudFormation, AWS Serverless Application Model (AWS SAM) et Terraform.

Vous êtes facturé pour le temps d'exécution que l'extension consomme (par incréments de 1 ms). Aucun coût n'est facturé pour l'installation de vos propres extensions. Pour obtenir des informations sur la tarification des extensions, consultez Tarification AWS Lambda. Pour obtenir des informations sur la tarification des extensions de partenaires, consultez les sites web de ces partenaires. Consultez the section called "Partenaires des extensions" pour obtenir la liste des extensions partenaires officielles.

Pour obtenir un didacticiel sur les extensions et la manière de les utiliser avec vos fonctions Lambda, consultez l'Atelier sur les extensions AWS Lambda.

Rubriques

- Environnement d'exécution
- Impact sur les performances et les ressources

- Autorisations
- Configuration des extensions Lambda
- Partenaires des extensions AWS Lambda
- Utilisation de l'API d'extensions Lambda pour créer des extensions
- Accès aux données de télémétrie en temps réel pour les extensions à l'aide de l'API de télémétrie

Environnement d'exécution

Lambda invoque votre fonction dans un environnement d'exécution qui fournit un environnement d'exécution sécurisé et isolé. L'environnement d'exécution gère les ressources requises pour exécuter votre fonction et prend en charge le cycle de vie pour l'exécution de la fonction et les extensions associées à votre fonction.

Le cycle de vie de l'environnement d'exécution comprend les phases suivantes :

- **`Init`** : au cours de cette phase, Lambda crée ou libère un environnement d'exécution avec les ressources configurées, télécharge le code pour la fonction et toutes les couches, initialise les extensions, initialise l'exécution et exécute le code d'initialisation de la fonction (code en dehors du gestionnaire principal). La phase `Init` se produit soit lors de la première invocation, soit avant invocations de fonctions si vous avez activé la simultanéité approvisionnée.

  La phase `Init` est fractionnée en trois sous-phases : `Extension init`, `Runtime init` et `Function init`. Ces sous-phases garantissent que toutes les extensions et l'exécution accomplissent leurs tâches de configuration avant l'exécution du code de la fonction.

  Lorsque Lambda SnapStart est activé, la phase `Init` se produit lorsque vous publiez une version de la fonction. Lambda enregistre un instantané de l'état de la mémoire et du disque de l'environnement d'exécution initialisé, fait persister l'instantané chiffré et le met en cache pour un accès à faible latence. Si vous avez un hook d'environnement d'exécution avant le point de contrôle, le code s'exécute alors à la fin de la phase `Init`.
- **`Restore`** (SnapStart uniquement) : lorsque vous invoquez pour la première fois une fonction SnapStart et à mesure que la fonction augmente, Lambda reprend les nouveaux environnements d'exécution à partir de l'instantané persistant au lieu d'initialiser la fonction à partir de zéro. Si vous avez un hook d'exécution après restauration, le code s'exécute à la fin de la phase `Restore`. Vous êtes facturé pour la durée des hooks d'exécution après restauration. L'exécution doit se charger et les hooks d'exécution après restauration doivent se terminer dans le délai imparti (10 secondes).

Sinon, vous obtiendrez une SnapStartTimeoutException. Lorsque la phase `Restore` se termine, Lambda invoque le gestionnaire de fonction (Phase d'invocation).

- **`Invoke`** : au cours de cette phase, Lambda invoque le gestionnaire de la fonction. Une fois l'exécution de la fonction terminée, Lambda se prépare à gérer une autre invocation de fonction.
- **`Shutdown`**: cette phase se déclenche si la fonction Lambda ne reçoit aucune invocation pendant un certain temps. Au cours de la phase `Shutdown`, Lambda arrête l'exécution, alerte les extensions pour les laisser s'arrêter proprement, puis supprime l'environnement. Lambda envoie à chaque extension un événement `Shutdown` indiquant que l'environnement est sur le point d'être arrêté.

Au cours de la phase `Init`, Lambda extrait des couches contenant des extensions dans le répertoire `/opt` dans l'environnement d'exécution. Lambda recherche des extensions dans le répertoire `/opt/extensions/`, interprète chaque fichier comme un fichier d'amorçage exécutable pour le lancement de l'extension, puis démarre toutes les extensions en parallèle.

Impact sur les performances et les ressources

La taille des extensions de votre fonction compte dans la limite de taille du package de déploiement. Pour une archive de fichiers .zip, la taille totale après décompression de la fonction et de toutes les extensions ne peut pas dépasser la limite de taille du package de déploiement fixée à 250 Mo.

Les extensions peuvent avoir un impact sur les performances de votre fonction car elles partagent des ressources telles que le processeur, la mémoire et le stockage. Par exemple, si une extension effectue des opérations intensives de calcul, il se peut que vous observiez une augmentation de la durée d'exécution de la fonction.

Chaque extension doit terminer son initialisation avant que Lambda invoque la fonction. Par conséquent, une extension qui consomme un temps d'initialisation important peut augmenter la latence de l'invocation de la fonction.

Pour mesurer le temps supplémentaire que prend l'extension après l'exécution de la fonction, vous pouvez utiliser la métrique de fonction `PostRuntimeExtensionsDuration`. Pour mesurer l'augmentation de la mémoire utilisée, vous pouvez utiliser la métrique `MaxMemoryUsed`. Pour comprendre l'impact d'une extension spécifique, vous pouvez exécuter différentes versions de vos fonctions côte à côte.

> **Note**
>
> La métrique `MaxMemoryUsed` est l'une des métriques collectées par Lambda Insights et non une métrique native Lambda.

Autorisations

Les extensions ont accès aux mêmes ressources que les fonctions. Étant donné que les extensions sont exécutées dans le même environnement que la fonction, les autorisations sont partagées entre la fonction et l'extension.

Pour une archive de fichiers .zip, vous pouvez créer un modèle CloudFormation, afin de simplifier la tâche consistant à attacher la même configuration d'extension, y compris les autorisations Gestion des identités et des accès AWS (IAM), à plusieurs fonctions.

Configuration des extensions Lambda

Configuration des extensions (archive de fichiers .zip)

Vous pouvez ajouter une extension à votre fonction en tant que couche Lambda. L'utilisation de couches vous permet de partager des extensions au sein de votre organisation ou avec l'ensemble de la communauté des développeurs Lambda. Vous pouvez ajouter une ou plusieurs extensions à une couche. Vous pouvez enregistrer jusqu'à 10 extensions pour une fonction.

Ajoutez l'extension à votre fonction en utilisant la même méthode que pour n'importe quelle couche. Pour de plus amples informations, consultez Couches Lambda.

Ajout d'une extension à votre fonction (console)

1. Ouvrez la page Functions (Fonctions) de la console Lambda.
2. Choisissez une fonction.
3. Sélectionnez l'onglet Code s'il n'est pas déjà sélectionné.
4. Sous Layers (Couches), sélectionnez Edit (Modifier).
5. Pour Choose a layer (Choisir une couche), sélectionnez Specify an ARN (Spécifier un nom ARN).
6. Pour Specify an ARN (Spécifier un ARN), entrez l’Amazon Resource Name (ARN) d'une couche d'extension.
7. Choisissez Ajouter.

Utilisation des extensions dans les images de conteneur

Vous pouvez ajouter des extensions à votre image de conteneur. Le paramètre d'image de conteneur ENTRYPOINT détermine le processus principal de la fonction. Configurez le paramètre ENTRYPOINT dans le Dockerfile ou en tant que remplacement dans la configuration de la fonction.

Vous pouvez exécuter plusieurs traitements au sein d'un conteneur. Lambda gère le cycle de vie du traitement principal et tous les traitements supplémentaires. Lambda utilise l'API des extensions pour gérer le cycle de vie des extensions.

Exemple : ajout d'une extension externe

Une extension externe s'exécute dans un processus distinct de la fonction Lambda. Lambda démarre un traitement pour chaque extension dans le répertoire `/opt/extensions/`. Lambda utilise l'API

d'extensions pour gérer le cycle de vie des extensions. Une fois l'exécution de la fonction terminée, Lambda envoie un événement `Shutdown` à chaque extension externe.

Example d'ajouter une extension externe à une image de base Python

```
FROM public.ecr.aws/lambda/python:3.11

# Copy and install the app
COPY /app /app
WORKDIR /app
RUN pip install -r requirements.txt

# Add an extension from the local directory into /opt/extensions
ADD my-extension.zip /opt/extensions
CMD python ./my-function.py
```

Étapes suivantes

Pour de plus amples informations sur les extensions, nous vous recommandons les ressources suivantes :

- Pour consulter un exemple pratique de base, consultez Building Extensions for AWS Lambda sur le blog AWS Compute.
- Pour plus d'informations sur les extensions fournies par les partenaires AWS Lambda, consultez Introducing AWS Lambda Extensions sur le blog AWS Compute.
- Pour consulter les exemples d'extensions disponibles et scripts encapsuleurs, reportez-vous à Extensions AWS Lambda sur le référentiel d'exemples AWS GitHub.

Partenaires des extensions AWS Lambda

AWS Lambda s'est associé à plusieurs entités tierces pour fournir des extensions à intégrer à vos fonctions Lambda. La liste suivante détaille les extensions tierces qui sont prêtes à être utilisées à tout moment.

- AppDynamics : fournit une instrumentation automatique des fonctions Node.js ou Python Lambda, offrant une visibilité et des alertes sur les performances des fonctions.
- Axiom : fournit des tableaux de bord pour surveiller les performances de la fonction Lambda et les métriques agrégées au niveau du système.
- Datadog : offre une visibilité complète en temps réel pour vos applications sans serveur grâce à l'utilisation de métriques, de traces et de journaux.
- Dynatrace : offre une visibilité sur les traces et les métriques, et exploite l'IA pour la détection automatisée des erreurs et l'analyse des causes profondes sur l'ensemble de la pile d'applications.
- Elastic : offre une surveillance de la performance des applications (APM) pour identifier et résoudre la cause profonde des problèmes à l'aide de traces, de métriques et de journaux corrélés.
- Epsagon – écoute les événements d'invocation, stocke les traces et les envoie en parallèle aux exécutions de fonctions Lambda.
- Fastly– protège vos fonctions Lambda contre les activités suspectes, telles que les attaques de type injection, l'usurpation de compte par surcharge d'informations d'identification, les robots malveillants et les extorsions d'API.
- HashiCorp Vault – gère les secrets et les met à la disposition des développeurs pour qu'ils les utilisent dans le code de fonction, sans rendre compte aux fonctions Vault.
- Honeycomb – outil d'observabilité pour déboguer votre pile d'applications.
- Lumigo – profile les appels de fonctions Lambda et collecte des métriques pour résoudre les problèmes dans les environnements sans serveur et de microservice.
- New Relic – exécutée avec les fonctions Lambda, collecte, améliore et transporte automatiquement la télémétrie vers la plateforme d'observabilité unifiée de New Relic.
- Sedai – une plateforme autonome de gestion du cloud, alimentée par l'IA/ML, qui offre une optimisation continue aux équipes d'exploitation du cloud afin de maximiser les économies, les performances et la disponibilité du cloud à grande échelle.
- Sentry – diagnostique, corrige et optimise les performances des fonctions Lambda.
- Site24x7 – permet une observabilité en temps réel dans vos environnements Lambda.

- Splunk – collecte des métriques haute résolution à faible latence pour une surveillance efficiente et efficace des fonctions Lambda.
- Sumo Logic – offre une visibilité sur l'état et les performances des applications sans serveur.
- Salt Security : simplifie la gouvernance de la posture des API et la sécurité des API pour les fonctions Lambda grâce à une configuration automatisée et à une prise en charge de divers environnements d'exécution.

AWSExtensions gérées

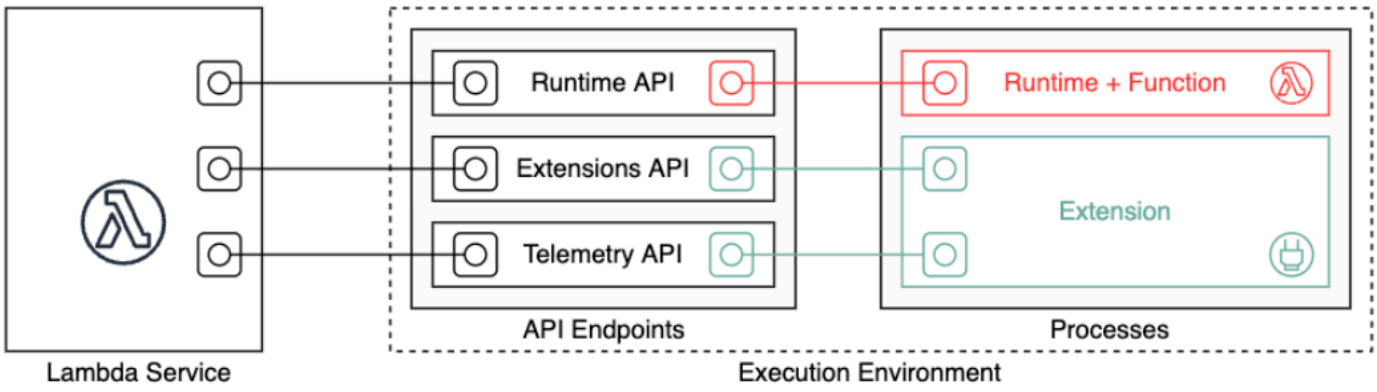
AWS fournit ses propres extensions gérées, notamment :

- AWS AppConfig : utilise des indicateurs de fonction et des données dynamiques pour mettre à jour vos fonctions Lambda. Vous pouvez également utiliser cette extension pour mettre à jour d'autres configurations dynamiques, telles que la limitation et le réglage des opérations.
- Amazon CodeGuru Profiler : améliore les performances des applications et réduit les coûts en identifiant la ligne de code la plus coûteuse d'une application et en fournissant des recommandations pour améliorer le code.
- CloudWatch Lambda Insights : surveille, dépanne et optimise les performances de vos fonctions Lambda grâce aux tableaux de bord automatisés.
- AWS Distro for Open Telemetry (ADOT) – Permet aux fonctions d'envoyer des données de traçage aux services de surveillance AWS tels que AWS X-Ray et vers des destinations prenant en charge OpenTelemetry, telles que Honeycomb et Lightstep.
- Paramètres et secrets AWS : récupère en toute sécurité les paramètres de AWS Systems Manager Parameter Store et les secrets de AWS Secrets Manager dans les fonctions Lambda.

Pour plus d'exemples d'extensions supplémentaires et de projets de démonstration, voir Extensions AWS Lambda.

Utilisation de l'API d'extensions Lambda pour créer des extensions

Les auteurs de fonctions Lambda utilisent des extensions pour intégrer Lambda avec leurs outils de prédilection en matière de surveillance, d'observabilité, de sécurité et de gouvernance. Les auteurs de fonctions peuvent utiliser des extensions issues de AWS projets open source, de AWS partenaires et de projets open source. Pour plus d'informations sur l'utilisation des extensions, consultez la section Présentation des AWS Lambda extensions sur le blog AWS Compute. Cette section décrit comment utiliser l'API Extensions Lambda, le cycle de vie de l'environnement d'exécution Lambda et la référence de l'API Extensions Lambda.



En tant qu'auteur d'extension, vous pouvez utiliser l'API d'extensions Lambda pour opérer une intégration profonde à l'environnement d'exécution Lambda. Votre extension peut enregistrer les événements du cycle de vie de la fonction et de l'environnement d'exécution. En réponse à ces événements, vous pouvez démarrer de nouveaux processus, exécuter une logique, ainsi que contrôler et orienter toutes les phases du cycle de vie Lambda : initialisation, appel et arrêt. En outre, vous pouvez utiliser l'API Runtime Logs pour recevoir un flux de journaux.

Une extension s'exécute en tant que processus indépendant dans l'environnement d'exécution et peut continuer de s'exécuter une fois l'appel de fonction entièrement traité. Étant donné que les extensions s'exécutent en tant que processus, vous pouvez les écrire dans un langage différent de celui de la fonction. Nous vous recommandons d'implémenter des extensions à l'aide d'un langage compilé. Dans ce cas, l'extension est un fichier binaire autonome compatible avec les environnements d'exécution pris en charge. Tous les Environnements d'exécution (runtimes) Lambda prennent en charge des extensions. Si vous utilisez un langage non compilé, assurez-vous d'inclure un environnement d'exécution compatible dans l'extension.

Lambda prend également en charge les extensions internes. Une extension interne s'exécute comme un thread séparé dans le processus d'exécution. L'exécution démarre et arrête l'extension interne. Un

autre mode d'intégration à l'environnement Lambda consiste à utiliser des variables d'environnement spécifiques au langage et des scripts encapsuleurs. Vous pouvez utiliser ces paramètres pour configurer l'environnement d'exécution et modifier le comportement de démarrage du processus d'exécution.

Vous pouvez ajouter des extensions à une fonction de deux manières. Pour une fonction déployée en tant qu'archive de fichier .zip, vous déployez votre extension en tant que couche. Pour une fonction définie comme une image de conteneur, vous ajoutez les extensions à cette dernière.

> **Note**
>
> Pour des exemples d'extensions et de scripts wrapper, voir AWS Lambda Extensions dans le GitHub référentiel AWS Samples.

Rubriques

- Cycle de vie d'un environnement d'exécution Lambda
- Référence d'API d'extensions

Cycle de vie d'un environnement d'exécution Lambda

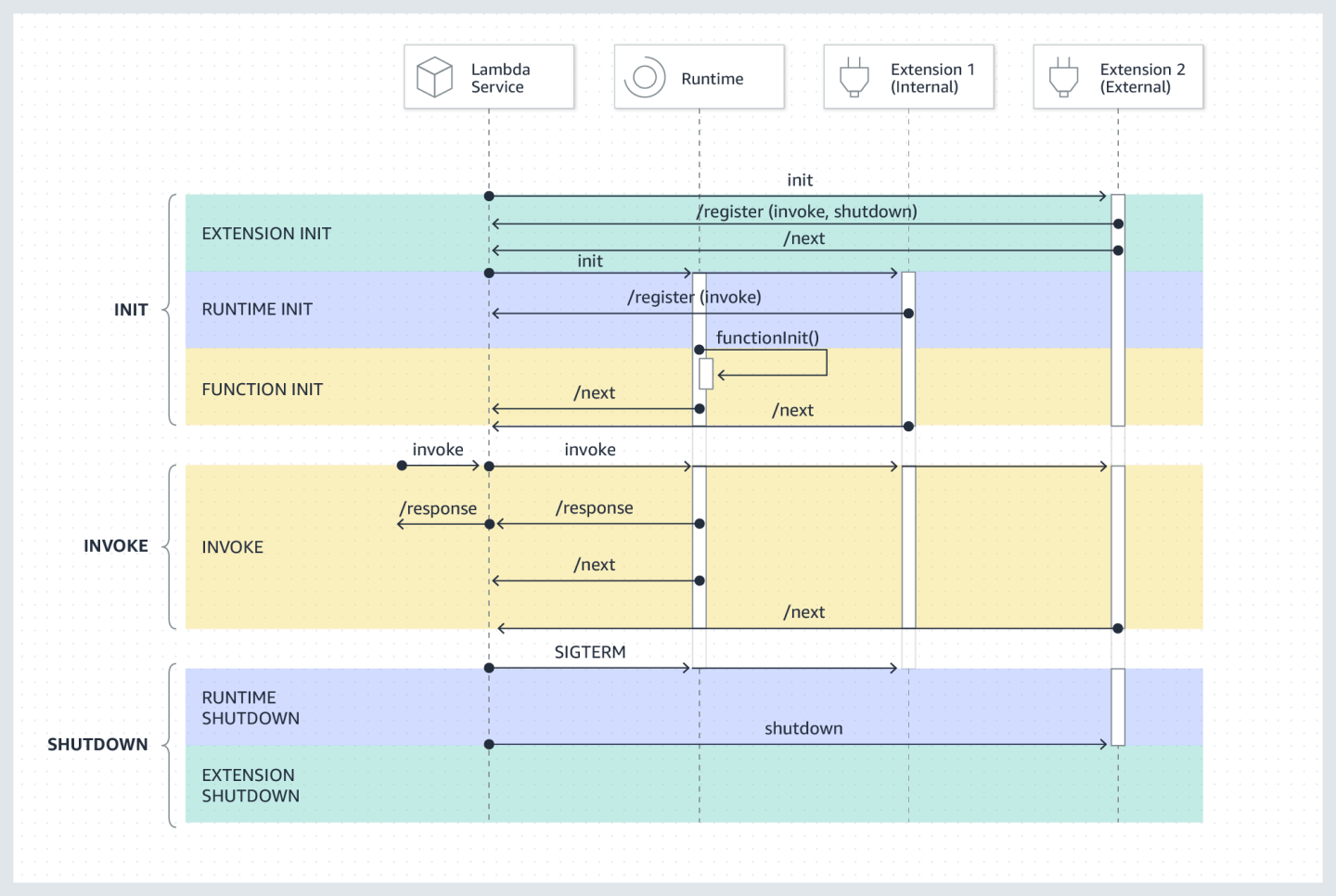
Le cycle de vie de l'environnement d'exécution comprend les phases suivantes :

- **`Init`** : au cours de cette phase, Lambda crée ou libère un environnement d'exécution avec les ressources configurées, télécharge le code pour la fonction et toutes les couches, initialise les extensions, initialise l'exécution et exécute le code d'initialisation de la fonction (code en dehors du gestionnaire principal). La phase `Init` se produit soit lors de la première invocation, soit avant invocations de fonctions si vous avez activé la simultanéité approvisionnée.

  La phase `Init` est fractionnée en trois sous-phases : `Extension init`, `Runtime init` et `Function init`. Ces sous-phases garantissent que toutes les extensions et l'exécution accomplissent leurs tâches de configuration avant l'exécution du code de la fonction.
- **`Invoke`** : au cours de cette phase, Lambda appelle le gestionnaire de la fonction. Une fois l'exécution de la fonction terminée, Lambda se prépare à gérer une autre invocation de fonction.
- **`Shutdown`**: cette phase se déclenche si la fonction Lambda ne reçoit aucune invocation pendant un certain temps. Au cours de la phase `Shutdown`, Lambda arrête l'exécution, alerte les extensions pour les laisser s'arrêter proprement, puis supprime l'environnement. Lambda envoie à

chaque extension un événement `Shutdown` indiquant que l'environnement est sur le point d'être arrêté.

Chaque phase commence par un événement que Lambda envoie au runtime et à toutes les extensions enregistrées. L'exécution et chaque extension signalent la fin de l'opération en envoyant une requête d'API `Next`. Lambda bloque l'environnement d'exécution lorsque l'exécution de chaque processus est terminée, et qu'il n'y a pas d'événement en attente.



Rubriques

- Phase d'initialisation
- Phase d'appel
- Phase d'arrêt
- Autorisations et configuration
- Gestion des défaillances

- Dépannage des extensions

Phase d'initialisation

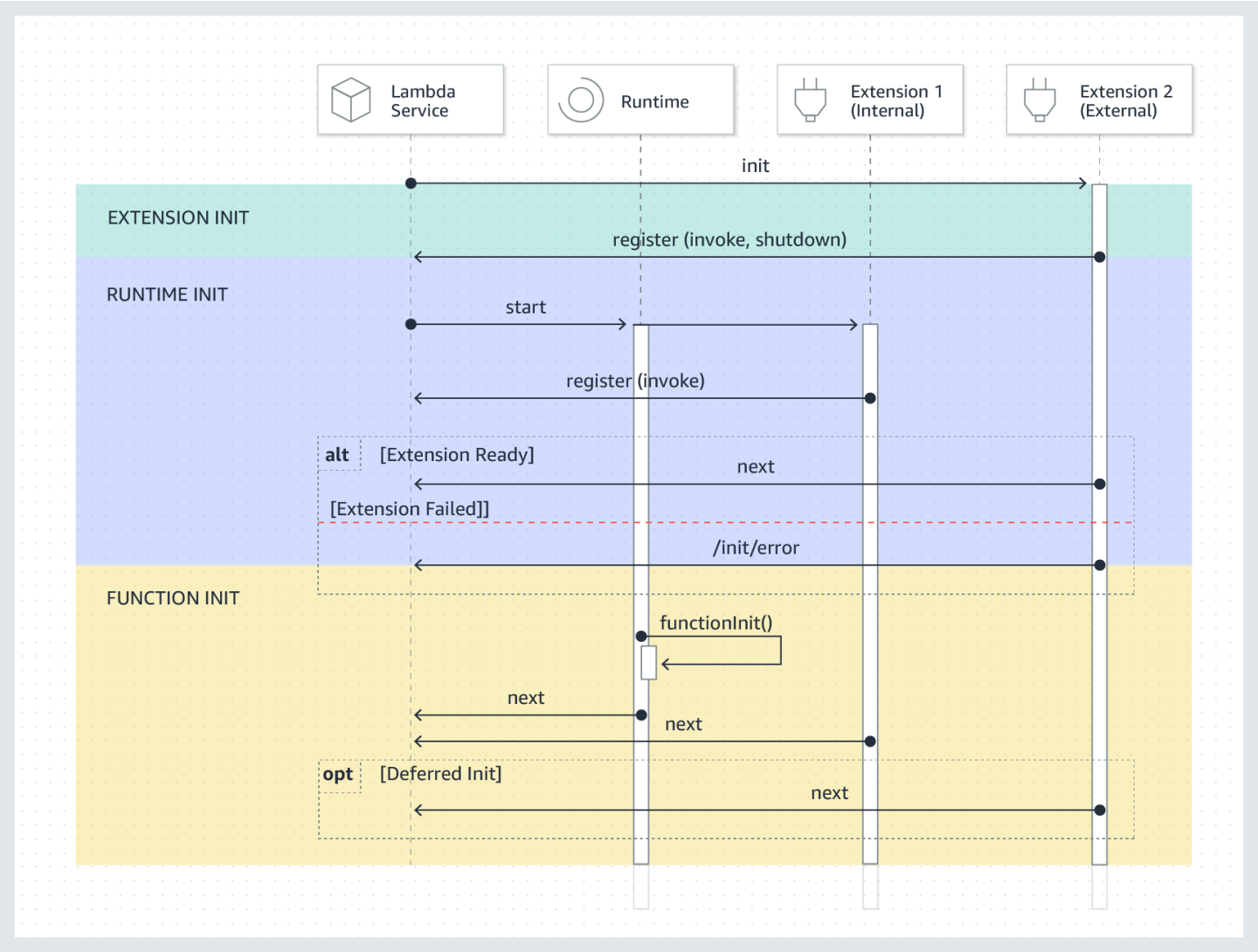
Au cours de la phase `Extension init`, chaque extension doit s'enregistrer auprès de Lambda pour recevoir des événements. Lambda utilise le nom de fichier complet de l'extension pour vérifier que la séquence d'amorçage de celle-ci est terminée. Par conséquent, chaque appel d'API `Register` doit inclure l'en-tête `Lambda-Extension-Name` avec le nom de fichier complet de l'extension.

Vous pouvez enregistrer jusqu'à 10 extensions pour une fonction. Cette limite est appliquée via l'appel d'API `Register`.

Après l'enregistrement de chaque extension, Lambda démarre la phase `Runtime init`. Le processus d'exécution appelle `functionInit` pour démarrer la phase `Function init`.

La phase `Init` se termine lorsque l'environnement d'exécution et chaque extension enregistrée indiquent la fin de l'opération en envoyant une demande d'API `Next`.

> **Note**
>
> Les extensions peuvent terminer leur initialisation à n'importe quel moment de la phase `Init`.



Phase d’appel

Lorsqu’une fonction Lambda est appelée en réponse à une requête d’API `Next`, Lambda envoie un événement `Invoke` à l’environnement d’exécution et à chaque extension enregistrée pour l’événement `Invoke`.

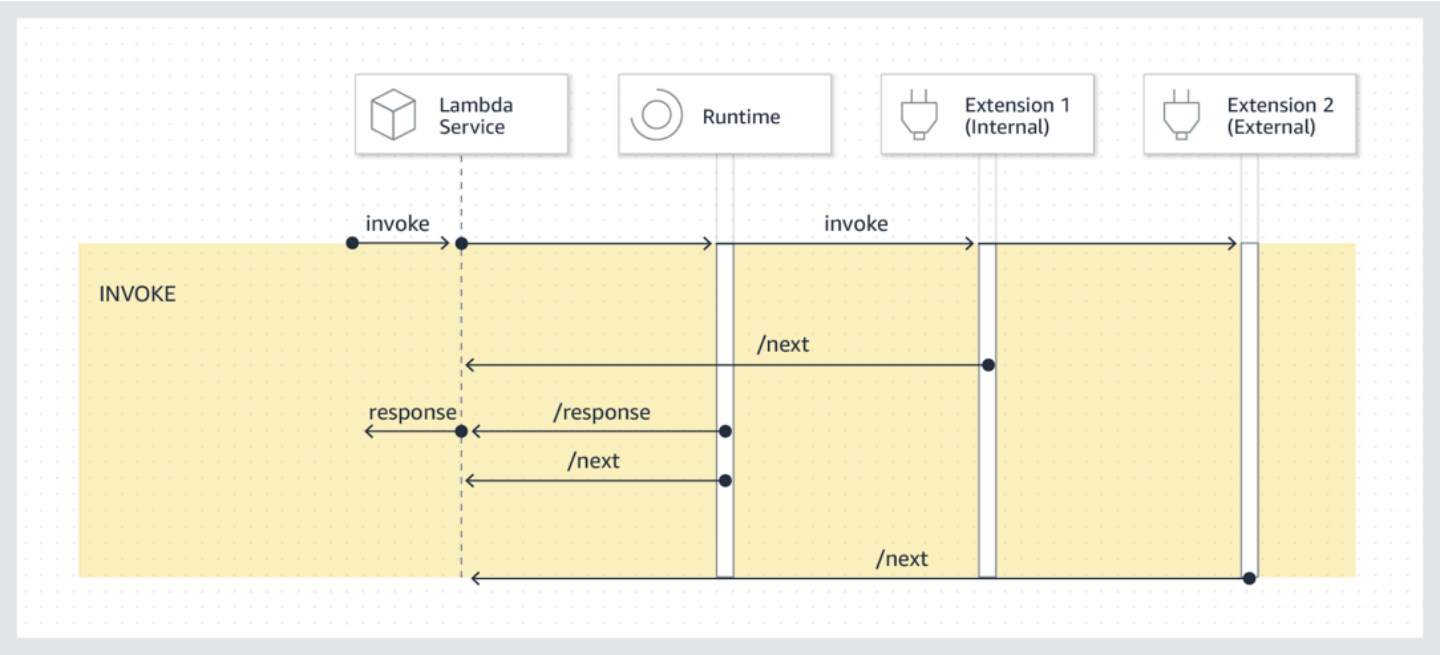
> **Note**
>
> Instances gérées Lambda : les extensions pour les fonctions des instances gérées Lambda ne peuvent pas être enregistrées pour l'événement. `Invoke` Comme les instances gérées Lambda prennent en charge les appels simultanés au sein d'un même environnement d'exécution, l'`Invoke`événement n'est pas pris en charge. Les extensions ne peuvent être

enregistrées que pour l'`Shutdown`événement. Si vous devez suivre le début et la fin des invocations, utilisez l'événement de `platform.report` plateforme via l'API de télémétrie.

Pendant l'appel, les extensions externes s'exécutent en parallèle avec la fonction. Elles continuent également à s'exécuter après la fin de la fonction. Cela vous permet de capturer des informations de diagnostic ou d'envoyer des journaux, des métriques et des suivis à l'emplacement de votre choix.

Après réception de la réponse de fonction de l'environnement d'exécution, Lambda renvoie celle-ci au client, même si les extensions sont toujours en cours d'exécution.

La phase `Invoke` prend fin lorsque l'environnement d'exécution et toutes les extensions signalent qu'ils ont terminé en envoyant une demande d'API `Next`.



Charge utile de l'événement : l'événement envoyé à l'environnement d'exécution (et à la fonction Lambda) transporte la totalité de la requête, les en-têtes (tels que `RequestId`) et la charge utile. L'événement envoyé à chaque extension contient des métadonnées décrivant le contenu de l'événement. Cet événement de cycle de vie inclut le type de l'événement, l'heure à laquelle la fonction expire (`deadlineMs`), la `requestId`, l'Amazon Resource Name (ARN) de la fonction appelée et les en-têtes de suivi.

Les extensions qui souhaitent accéder au corps de l'événement de la fonction peuvent utiliser un kit SDK interne à l'environnement d'exécution qui communique avec l'extension. Les développeurs

de fonctions utilisent le kit SDK interne à l'environnement d'exécution pour envoyer la charge utile à l'extension lorsque la fonction est appelée.

Voici un exemple de charge utile :

```
{
    "eventType": "INVOKE",
    "deadlineMs": 676051,
    "requestId": "3da1f2dc-3222-475e-9205-e2e6c6318895",
    "invokedFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:ExtensionTest",
    "tracing": {
        "type": "X-Amzn-Trace-Id",
        "value":
 "Root=1-5f35ae12-0c0fec141ab77a00bc047aa2;Parent=2be948a625588e32;Sampled=1"
    }
 }
```

Limite de durée : le paramètre d'expiration de la fonction limite la durée de la phase `Invoke` entière. Par exemple, si vous définissez le délai d'expiration de la fonction sur 360 secondes, la fonction et toutes les extensions doivent être terminées dans un délai de 360 secondes. Notez qu'il n'y a pas de phase post-invocation indépendante. La durée correspond à la durée totale nécessaire pour que votre environnement d'exécution et toutes vos invocations d'extensions se terminent. Elle n'est calculée que lorsque l'exécution de la fonction et de toutes les extensions est terminée.

Impact sur les performances et surcharge d'extension : les extensions peuvent avoir un impact sur les performances des fonctions. En tant qu'auteur d'extension, vous contrôlez l'impact de votre extension sur les performances. Par exemple, si votre extension effectue des opérations intensives de calcul, la durée de la fonction augmente, car le code de l'extension et de votre fonction partage les mêmes ressources d'UC. En outre, si votre extension effectue des opérations importantes après la fin de l'appel de fonction, la durée de la fonction augmente, car la phase `Invoke` continue jusqu'à ce que toutes les extensions signalent qu'elles sont terminées.

> **Note**
>
> Lambda alloue une puissance de processeur proportionnelle au paramètre de mémoire de la fonction. La durée d'exécution et d'initialisation peut augmenter avec des paramètres de mémoire inférieurs, car les processus de fonction et d'extensions sont en concurrence pour

les mêmes ressources de processeur. Pour réduire la durée d'exécution et d'initialisation, essayez d'augmenter le paramètre de mémoire.

Pour vous aider à déterminer l'impact sur les performances des extensions dans la phase `Invoke`, Lambda génère la métrique `PostRuntimeExtensionsDuration`. Cette métrique mesure le temps cumulé qui s'écoule entre la demande d'API `Next` de l'environnement d'exécution et la dernière demande d'API `Next` d'une extension. La métrique `MaxMemoryUsed` permet de mesurer l'augmentation de la mémoire utilisée. Pour de plus amples informations sur les métriques de fonction, veuillez consulter Utilisation des métriques CloudWatch avec Lambda.

Les développeurs de fonctions peuvent exécuter différentes versions de leurs fonctions côte à côte pour comprendre l'impact d'une extension spécifique. Nous recommandons aux auteurs d'extension de publier la consommation de ressources prévue de manière à aider les développeurs de fonction dans leur choix de l'extension appropriée.

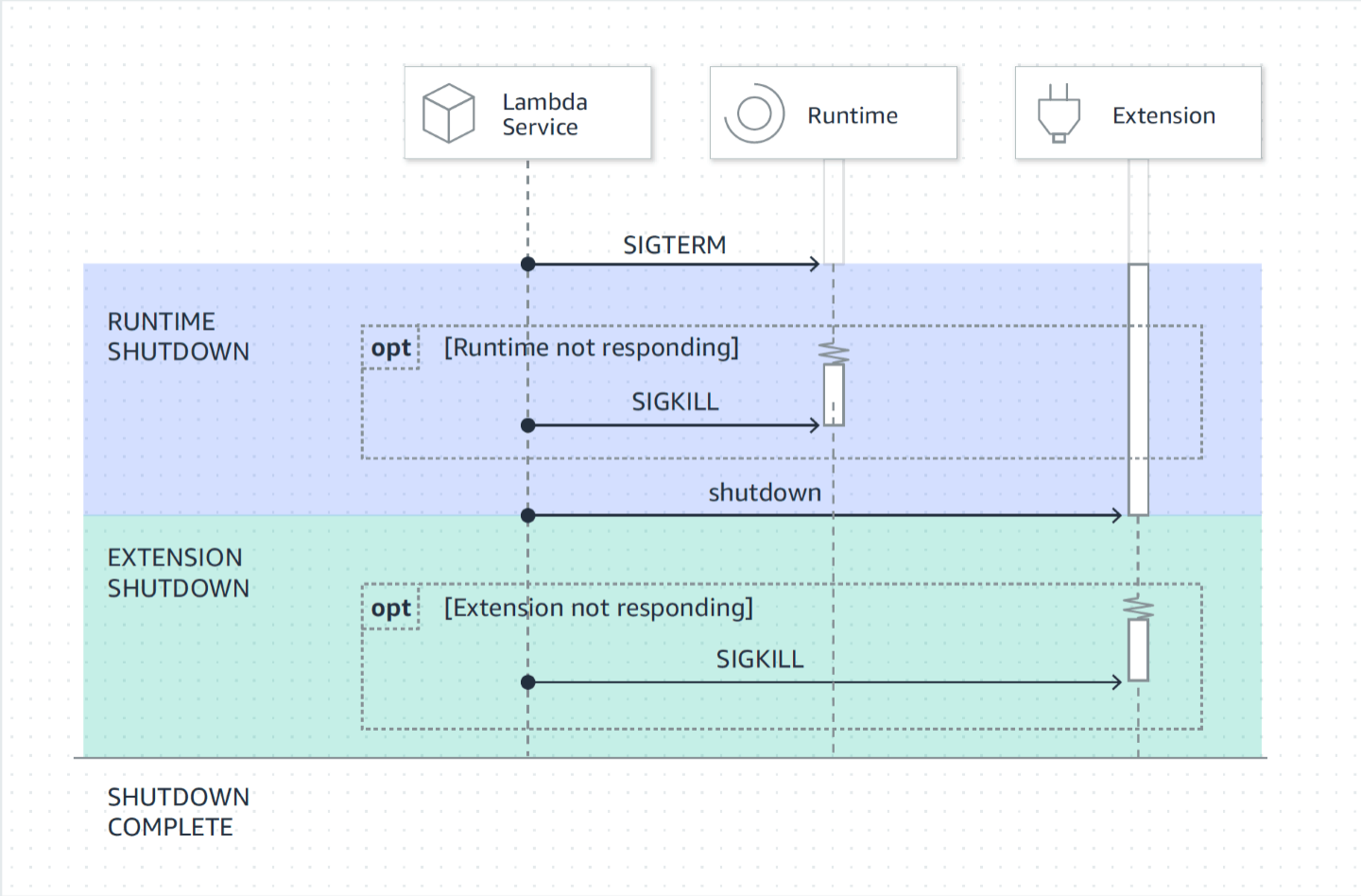
Phase d'arrêt

Quand Lambda est sur le point d'arrêter l'exécution, il envoie un `Shutdown` à chaque extension externe enregistrée. Les extensions peuvent utiliser ce temps pour les tâches de nettoyage final. L'événement `Shutdown` est envoyé en réponse à une demande d'API `Next`.

Limite de durée : la durée maximale de la phase `Shutdown` dépend de la configuration des extensions enregistrées :

- 0 ms : fonction sans extension enregistrée
- 500 ms : fonction avec une extension interne enregistrée.
- 2 000 ms : fonction avec une ou plusieurs extensions externes enregistrées.

Si l'environnement d'exécution ou une extension ne répondent pas à l'événement `Shutdown` dans cette limite de temps, Lambda met fin au processus à l'aide d'un signal `SIGKILL`.



Charge utile d'événement : l'événement `Shutdown` contient la raison de l'arrêt et le temps restant en millisecondes.

`shutdownReason` contient les éléments suivants :

- SPINDOWN – Arrêt normal
- TIMEOUT – Limite de durée dépassée
- ÉCHEC : condition d'erreur, telle qu'un événement `out-of-memory`

```
{
  "eventType": "SHUTDOWN",
  "shutdownReason": "reason for shutdown",
  "deadlineMs": "the time and date that the function times out in Unix time
 milliseconds"
}
```

Autorisations et configuration

Les extensions s'exécutent dans le même environnement d'exécution que la fonction Lambda. Les extensions partagent également des ressources avec la fonction, telles que le processeur, la mémoire et le disque de stockage `/tmp`. En outre, les extensions utilisent le même rôle Gestion des identités et des accès AWS (IAM) et le même contexte de sécurité que la fonction.

Autorisations d'accès au système de fichiers et au réseau : les extensions s'exécutent dans le même espace de noms de système de fichiers et de nom de réseau que l'environnement d'exécution de la fonction. Cela signifie que les extensions doivent être compatibles avec le système d'exploitation associé. Si une extension nécessite des règles supplémentaires de trafic réseau sortant, vous devez appliquer celles-ci à la configuration de la fonction.

> **Note**
>
> Étant donné que le répertoire du code de la fonction est en lecture seule, les extensions ne peuvent pas modifier le code de la fonction.

Variables d'environnement : les extensions peuvent accéder aux variables d'environnement de la fonction, à l'exception des variables suivantes spécifiques au processus d'environnement d'exécution :

- `AWS_EXECUTION_ENV`
- `AWS_LAMBDA_LOG_GROUP_NAME`
- `AWS_LAMBDA_LOG_STREAM_NAME`
- `AWS_LAMBDA_MAX_CONCURRENCY`(Instances gérées par Lambda uniquement)
- `AWS_XRAY_CONTEXT_MISSING`
- `AWS_XRAY_DAEMON_ADDRESS`
- `LAMBDA_RUNTIME_DIR`
- `LAMBDA_TASK_ROOT`
- `_AWS_XRAY_DAEMON_ADDRESS`
- `_AWS_XRAY_DAEMON_PORT`

- `_HANDLER`

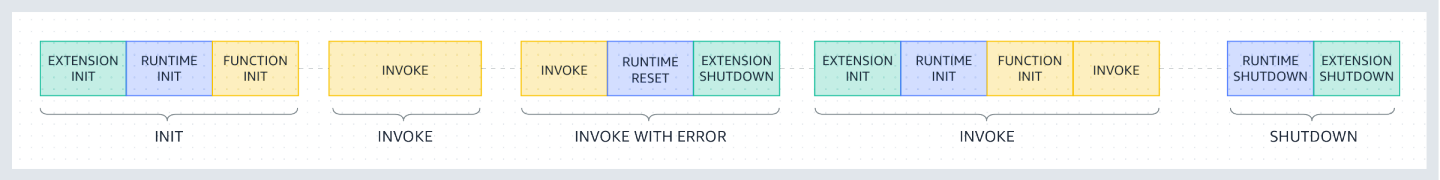
> **Note**
>
> Détection des instances gérées Lambda : les extensions peuvent vérifier la variable d'`AWS_LAMBDA_INITIALIZATION_TYPE`environnement pour déterminer si elles s'exécutent sur des instances gérées Lambda plutôt que sur des fonctions Lambda (par défaut). Il s'agit de la méthode recommandée pour que les extensions adaptent leur comportement en fonction du type d'environnement d'exécution.

Gestion des défaillances

Échecs d'initialisation : si une extension échoue, Lambda redémarre l'environnement d'exécution pour assurer un comportement cohérent et encourager un échec rapide des extensions. En outre, pour certains clients, les extensions doivent répondre à des besoins stratégiques tels que la journalisation, la sécurité, la gouvernance et la collecte de télémétrie.

Échecs d'appel (par exemple, manque de mémoire, expiration de fonction) : les extensions partageant des ressources avec l'environnement d'exécution, elles sont affectées par l'épuisement de la mémoire. Lorsque l'environnement d'exécution échoue, toutes les extensions et l'environnement d'exécution lui-même participent à la phase `Shutdown`. En outre, l'environnement d'exécution est redémarré soit automatiquement dans le cadre de l'appel actuel, soit via un mécanisme de réinitialisation différée.

En cas d'échec (par exemple, dépassement de délai d'attente de fonction ou erreur d'exécution) pendant la phase `Invoke`, le service Lambda effectue une réinitialisation. La réinitialisation se comporte comme un événement `Shutdown`. Lambda commence par arrêter l'environnement d'exécution, puis envoie un événement `Shutdown` à chaque extension externe enregistrée. L'événement comprend le motif de l'arrêt. Si cet environnement est utilisé pour un nouvel appel, l'extension et l'environnement d'exécution sont réinitialisés dans le cadre de l'appel suivant.



Pour une explication plus détaillée du diagramme précédent, consultez Échecs pendant la phase d'invocation.

Journaux des extensions : Lambda envoie le résultat du journal des extensions à CloudWatch Logs. Lambda génère également un événement de journal supplémentaire pour chaque extension pendant la phase `Init`. L'événement de journal enregistre le nom et la préférence d'enregistrement (événement, configuration) en cas de succès, ou la raison de l'échec le cas échéant.

Dépannage des extensions

- Si une demande `Register` échoue, assurez-vous que l'en-tête `Lambda-Extension-Name` de l'appel d'API `Register` contient le nom de fichier complet de l'extension.
- Si la demande `Register` échoue pour une extension interne, assurez-vous que la demande n'est pas enregistrée pour l'événement `Shutdown`.

Référence d'API d'extensions

La spécification OpenAPI pour l'API d'extensions version 2020-01-01 est disponible ici : extensions-api.zip

Vous pouvez extraire la valeur du point de terminaison d'API à partir de la variable d'environnement `AWS_LAMBDA_RUNTIME_API`. Pour envoyer une demande `Register`, utilisez le préfixe `2020-01-01/` avant chaque chemin d'API. Exemples :

```
http://${AWS_LAMBDA_RUNTIME_API}/2020-01-01/extension/register
```

Méthodes d'API

- Enregistrement
- Suivant
- Erreur d'initiation
- Erreur de sortie

Enregistrement

Au cours de la phase `Extension init`, chaque extension doit s'enregistrer auprès de Lambda pour recevoir les événements. Lambda utilise le nom de fichier complet de l'extension pour vérifier que la séquence d'amorçage de celle-ci est terminée. Par conséquent, chaque appel d'API `Register` doit inclure l'en-tête `Lambda-Extension-Name` avec le nom de fichier complet de l'extension.

Les extensions internes sont lancées et arrêtées par le processus d'environnement d'exécution, de sorte qu'elles ne peuvent pas être enregistrées pour l'événement `Shutdown`.

Chemin – `/extension/register`

Méthode – POST

En-têtes de demandes

- `Lambda-Extension-Name` : nom de fichier complet de l'extension. Obligatoire : oui. Type : chaîne.
- `Lambda-Extension-Accept-Feature` – Utilisez ceci pour spécifier les fonctionnalités optionnelles d'Extensions pendant l'enregistrement. Requis : non. Type : chaîne séparée par des virgules. Fonctions disponibles pour spécifier en utilisant ce paramètre :
  - `accountId` – Si elle est spécifiée, la réponse d'enregistrement de l'extension contiendra l'ID du compte associé à la fonction Lambda pour laquelle vous enregistrez l'extension.

Paramètres du corps de la demande

- `events` : tableau des événements pour lesquels s'enregistrer. Requis : non. Type : tableau de chaînes. Chaînes valides : `INVOKE`, `SHUTDOWN`.

> **Note**
>
> Instances gérées Lambda : les extensions pour les fonctions des instances gérées Lambda ne peuvent être enregistrées que pour l'événement. `SHUTDOWN` Toute tentative d'inscription à l'`INVOKE`événement provoquera une erreur. Cela est dû au fait que les instances gérées Lambda prennent en charge les appels simultanés au sein d'un environnement d'exécution unique.

En-têtes de réponse

- `Lambda-Extension-Identifier` : identifiant d'agent unique généré (chaîne UUID) requis pour toutes les requêtes subséquentes.

Codes de réponse

- 200 – Corps de la réponse contenant le nom de la fonction, la version de la fonction et le nom du gestionnaire.
- 400 – Demande erronée.
- 403 – Interdit
- 500 – Erreur de conteneur. État non récupérable. L'extension doit se fermer rapidement.

Example Exemple de corps de la demande

```
{
    'events': [ 'INVOKE', 'SHUTDOWN']
}
```

Example Exemple de corps de la réponse

```
{
    "functionName": "helloWorld",
    "functionVersion": "$LATEST",
    "handler": "lambda_function.lambda_handler"
}
```

Example Exemple de corps de réponse avec la fonction optionnelle accountId

```
{
    "functionName": "helloWorld",
    "functionVersion": "$LATEST",
    "handler": "lambda_function.lambda_handler",
    "accountId": "123456789012"
}
```

Suivant

Les extensions envoient une demande d'API `Next` pour recevoir l'événement suivant, qui peut être un événement `Invoke` ou un événement `Shutdown`. Le corps de la réponse contient la charge utile, sous la forme d'un document JSON contenant des données d'événement.

L'extension envoie une demande d'API `Next` pour signaler qu'elle est prête à recevoir de nouveaux événements. Il s'agit d'un appel bloquant.

Ne définissez pas de délai d'expiration sur l'appel GET, car l'extension peut être suspendue pendant un certain temps jusqu'à ce qu'il y ait un événement à retourner.

Chemin – `/extension/event/next`

Méthode – GET

En-têtes de demandes

- `Lambda-Extension-Identifier` : identifiant unique pour l'extension (chaîne UUID). Obligatoire : oui. Type : chaîne UUID.

En-têtes de réponse

- `Lambda-Extension-Event-Identifier` : identifiant unique pour l'événement (chaîne UUID).

Codes de réponse

- 200 : réponse contenant des informations sur l'événement suivant (`EventInvoke` ou `EventShutdown`).
- 403 – Interdit.
- 500 – Erreur de conteneur. État non récupérable. L'extension doit se fermer rapidement.

Erreur d'initiation

L'extension utilise cette méthode pour signaler une erreur d'initialisation à Lambda. Appelez-la lorsque l'extension ne parvient pas à s'initialiser après son enregistrement. Après que Lambda a reçu l'erreur, aucun autre appel d'API n'aboutit. L'extension doit quitter après avoir reçu la réponse de Lambda.

Chemin – `/extension/init/error`

Méthode – POST

En-têtes de demandes

- `Lambda-Extension-Identifier` : identifiant unique pour l'extension. Obligatoire : oui. Type : chaîne UUID.

- `Lambda-Extension-Function-Error-Type` – Type d'erreur rencontré par l'extension. Obligatoire : oui. Cet en-tête se compose d'une valeur de chaîne. Lambda accepte n'importe quelle chaîne, mais nous recommandons le format <category.reason>. Par exemple :
  - Prolongation. NoSuchHandler
  - Prolongation. APIKeyNotFound
  - Prolongation. ConfigInvalid
  - Prolongation. UnknownReason

Paramètres du corps de la demande

- `ErrorRequest` : informations sur l'erreur. Requis : non.

Ce champ est un objet JSON avec la structure suivante :

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

Notez que Lambda accepte n'importe quelle valeur pour `errorType`.

L'exemple suivant montre un message d'erreur de fonction Lambda indiquant que la fonction n'a pas pu analyser les données d'événement fournies dans l'invocation.

Example Erreur de fonction

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

Codes de réponse

- 202 – Accepté
- 400 – Demande erronée.
- 403 – Interdit

- 500 – Erreur de conteneur. État non récupérable. L'extension doit se fermer rapidement.

Erreur de sortie

L'extension utilise cette méthode pour signaler une erreur à Lambda avant de quitter. Appelez-la lorsque vous rencontrez une défaillance inattendue. Après que Lambda a reçu l'erreur, aucun autre appel d'API n'aboutit. L'extension doit quitter après avoir reçu la réponse de Lambda.

Chemin – `/extension/exit/error`

Méthode – POST

En-têtes de demandes

- `Lambda-Extension-Identifier` : identifiant unique pour l'extension. Obligatoire : oui. Type : chaîne UUID.
- `Lambda-Extension-Function-Error-Type` – Type d'erreur rencontré par l'extension. Obligatoire : oui. Cet en-tête se compose d'une valeur de chaîne. Lambda accepte n'importe quelle chaîne, mais nous recommandons le format <category.reason>. Par exemple :
  - Prolongation. NoSuchHandler
  - Prolongation. APIKeyNotFound
  - Prolongation. ConfigInvalid
  - Prolongation. UnknownReason

Paramètres du corps de la demande

- `ErrorRequest` : informations sur l'erreur. Requis : non.

Ce champ est un objet JSON avec la structure suivante :

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

Notez que Lambda accepte n'importe quelle valeur pour `errorType`.

L'exemple suivant montre un message d'erreur de fonction Lambda indiquant que la fonction n'a pas pu analyser les données d'événement fournies dans l'invocation.

Example Erreur de fonction

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

Codes de réponse

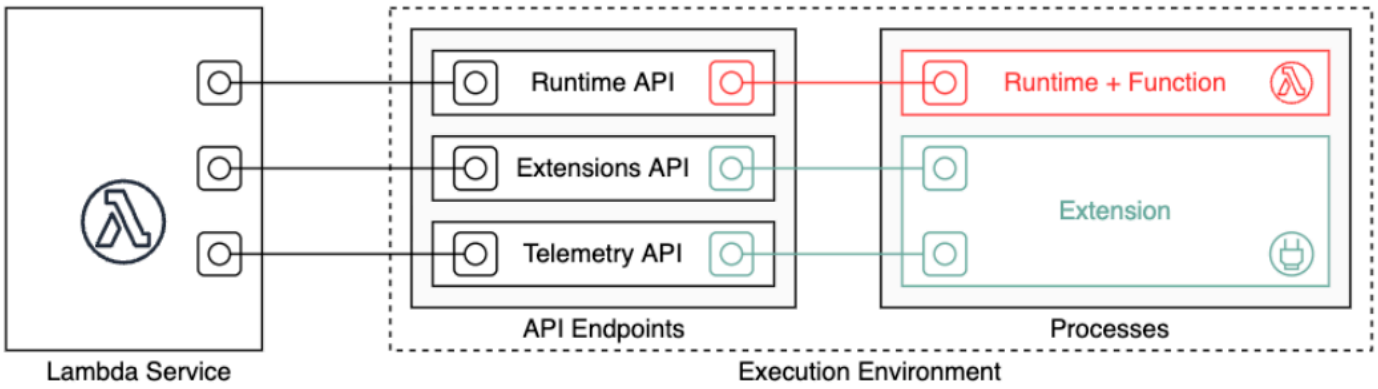
- 202 – Accepté
- 400 – Demande erronée.
- 403 – Interdit
- 500 – Erreur de conteneur. État non récupérable. L'extension doit se fermer rapidement.

Accès aux données de télémétrie en temps réel pour les extensions à l'aide de l'API de télémétrie

L'API de télémétrie active vos extensions pour recevoir directement des données de télémétrie de Lambda. Pendant l'initialisation et l'invocation des fonctions, Lambda capture automatiquement la télémétrie, notamment les journaux, les métriques de la plateforme et les traces de la plateforme. L'API de télémétrie active des extensions pour accéder à ces données de télémétrie directement depuis Lambda en temps quasi réel.

Dans l'environnement d'exécution Lambda, vous pouvez abonner vos extensions Lambda à des flux de télémétrie. Après la souscription, Lambda envoie automatiquement toutes les données de télémétrie à vos extensions. Vous pouvez ensuite traiter, filtrer et distribuer les données à votre destination préférée, telle qu'un compartiment Amazon Simple Storage Service (Amazon S3) ou un fournisseur d'outils d'observabilité tiers.

Le schéma suivant montre comment l'API d'extensions et l'API de télémétrie lient les extensions à Lambda depuis l'environnement d'exécution. De plus, l'API d'exécution connecte votre environnement d'exécution et votre fonction à Lambda.



> **Important**
>
> L'API de télémétrie Lambda remplace l'API de journaux Lambda. Bien que l'API de journaux reste entièrement fonctionnelle, nous vous recommandons d'utiliser uniquement l'API de télémétrie à l'avenir. Vous pouvez abonner votre extension à un flux de télémétrie en utilisant l'API de télémétrie ou l'API de journaux. Après vous être abonné à l'aide de l'une de ces API APIs, toute tentative de souscription à l'aide de l'autre API renvoie une erreur.

Exigence de version du schéma des instances gérées Lambda

Les instances gérées Lambda ne prennent en charge que la version du `2025-01-29` schéma de l'API de télémétrie. Lorsque vous vous abonnez à des flux de télémétrie pour les fonctions des instances gérées, vous devez les utiliser `"schemaVersion": "2025-01-29"` dans votre demande d'abonnement. L'utilisation de versions de schéma précédentes entraînera le rejet d'événements par Lambda.
La version du `2025-01-29` schéma est rétrocompatible et peut être utilisée à la fois avec les instances gérées Lambda et les fonctions Lambda (par défaut). Nous recommandons d'utiliser cette version pour toutes les nouvelles extensions afin de garantir la compatibilité entre les deux modèles de déploiement.

Les extensions peuvent utiliser l'API de télémétrie pour s'abonner à trois flux de télémétrie différents :

- Télémétrie de plateforme – Journaux, métriques et traces, qui décrivent les événements et les erreurs liés au cycle de vie de l'environnement d'exécution, au cycle de vie des extensions et aux appels de fonctions
- Journaux de fonction – Journaux personnalisés que le code de la fonction Lambda génère.
- Journaux d'extension – Journaux personnalisés générés par le code d'extension Lambda.

Note

Lambda envoie des journaux CloudWatch, des métriques et des traces à X-Ray (si vous avez activé le suivi), même si une extension s'abonne à des flux de télémétrie.

Sections

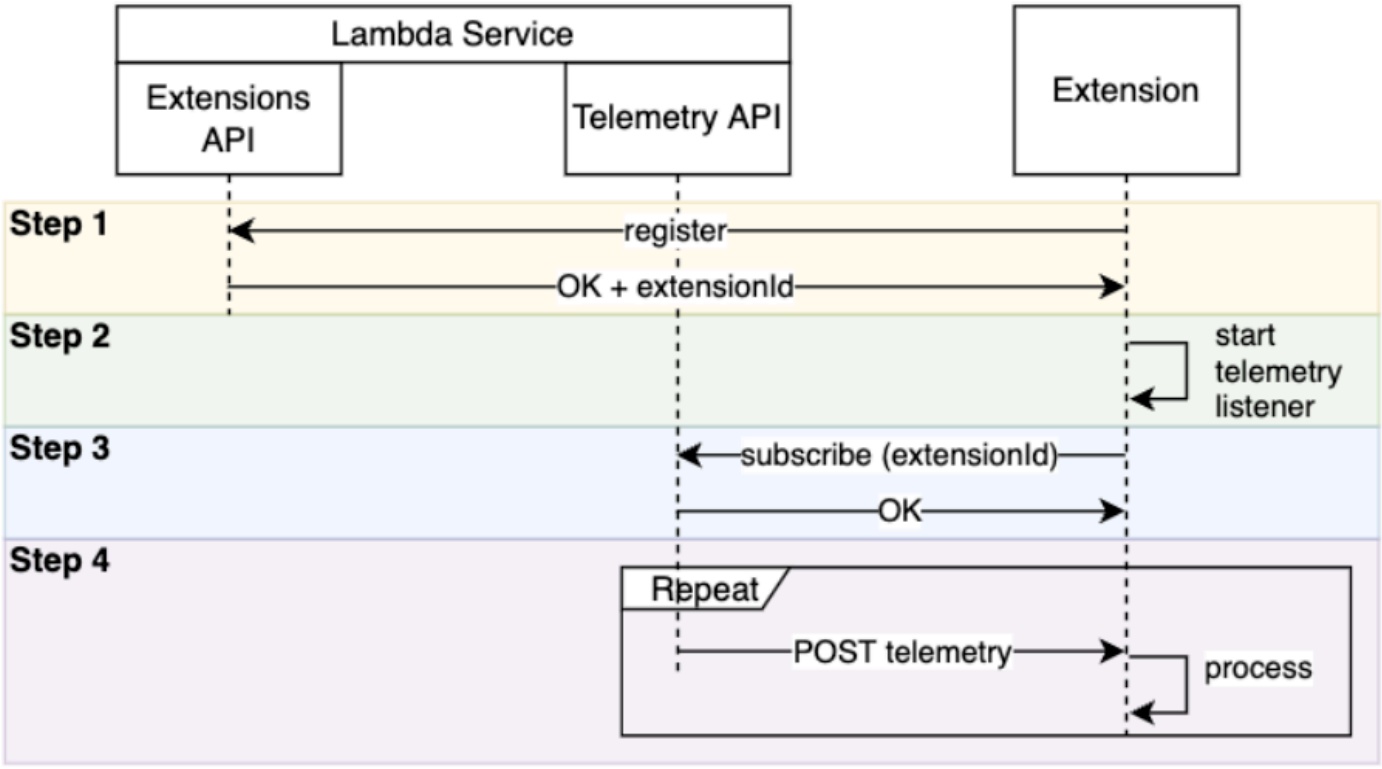
- Création d'extensions à l'aide de l'API de télémétrie
- Enregistrement de votre extension
- Création d'un écouteur de télémétrie
- Spécification d'un protocole de destination
- Configuration de l'utilisation de la mémoire et de la mise en mémoire tampon
- Envoi d'une demande d'abonnement à l'API de télémétrie
- Messages entrants de l'API de télémétrie

- Référence de l'API de télémétrie Lambda
- Référence du schéma Event de l'API de télémétrie Lambda
- Conversion des objets Event de l'API de télémétrie Lambda en OpenTelemetry Spans
- Utilisation de l'API Logs Lambda

Création d'extensions à l'aide de l'API de télémétrie

Les extensions Lambda s'exécutent comme des processus indépendants dans l'environnement d'exécution. Les extensions peuvent continuer à s'exécuter une fois l'invocation des fonctions terminée. Comme les extensions sont des processus séparés, vous pouvez les écrire dans un langage différent du code de la fonction. Nous recommandons d'écrire les extensions en utilisant un langage compilé tel que Golang ou Rust. De cette façon, l'extension est un binaire autonome qui peut être compatible avec tout environnement d'exécution pris en charge.

Le schéma suivant illustre un processus en quatre étapes pour créer une extension qui reçoit et traite des données de télémétrie à l'aide de l'API de télémétrie.



Voici chaque étape plus en détail :

1. Enregistrez votre extension à l'aide du the section called "API d'extensions". Cela vous fournit un `Lambda-Extension-Identifier`, dont vous aurez besoin dans les étapes suivantes. Pour plus d'informations sur la façon d'enregistrer votre extension, consultez the section called "Enregistrement de votre extension".
2. Créez un écouteur de télémétrie. Il peut s'agir d'un serveur HTTP ou TCP de base. Lambda utilise l'URI de l'écouteur de télémétrie pour envoyer des données de télémétrie à votre extension. Pour de plus amples informations, veuillez consulter the section called "Création d'un écouteur de télémétrie".
3. À l'aide de l'API d'abonnement de l'API de télémétrie, abonnez votre extension aux flux de télémétrie souhaités. Vous aurez besoin de l'URI de votre écouteur de télémétrie pour cette étape. Pour de plus amples informations, veuillez consulter the section called "Envoi d'une demande d'abonnement à l'API de télémétrie".
4. Obtenez les données de télémétrie de Lambda via l'écouteur de télémétrie. Vous pouvez effectuer tout traitement personnalisé de ces données, par exemple en les envoyant vers Amazon S3 ou vers un service d'observabilité externe.

> **Note**
>
> L'environnement d'exécution d'une fonction Lambda peut démarrer et s'arrêter plusieurs fois dans le cadre de son cycle de vie. En général, votre code d'extension s'exécute pendant les appels de la fonction, et aussi jusqu'à 2 secondes pendant la phase d'arrêt. Nous vous recommandons de regrouper la télémétrie au fur et à mesure qu'elle parvient à votre écouteur. Utilisez ensuite les événements du cycle de vie `Invoke` et `Shutdown` pour envoyer chaque lot vers les destinations souhaitées.

Enregistrement de votre extension

Avant de pouvoir vous abonner aux données de télémétrie, vous devez enregistrer votre extension Lambda. L'enregistrement a lieu pendant la phase d'initialisation de l'extension. L'exemple suivant montre une demande HTTP pour enregistrer une extension.

```
POST http://${AWS_LAMBDA_RUNTIME_API}/2020-01-01/extension/register
 Lambda-Extension-Name: lambda_extension_name
{
    'events': [ 'INVOKE', 'SHUTDOWN']
```

```
}
```

Si la demande réussit, l'abonné reçoit une réponse de succès HTTP 200. L'en-tête de réponse contient le `Lambda-Extension-Identifier`. Le corps de la réponse contient d'autres propriétés de la fonction.

```
HTTP/1.1 200 OK
Lambda-Extension-Identifier: a1b2c3d4-5678-90ab-cdef-EXAMPLE11111
{
    "functionName": "lambda_function",
    "functionVersion": "$LATEST",
    "handler": "lambda_handler",
    "accountId": "123456789012"
}
```

Pour plus d'informations, consultez le the section called "Référence d'API d'extensions".

Création d'un écouteur de télémétrie

Votre extension Lambda doit avoir un écouteur qui traite les demandes entrantes de l'API de télémétrie. Le code suivant présente un exemple de mise en œuvre d'un écouteur de télémétrie en Golang :

```
// Starts the server in a goroutine where the log events will be sent
func (s *TelemetryApiListener) Start() (string, error) {
 address := listenOnAddress()
 l.Info("[listener:Start] Starting on address", address)
 s.httpServer = &http.Server{Addr: address}
 http.HandleFunc("/", s.http_handler)
 go func() {
  err := s.httpServer.ListenAndServe()
  if err != http.ErrServerClosed {
   l.Error("[listener:goroutine] Unexpected stop on Http Server:", err)
   s.Shutdown()
  } else {
   l.Info("[listener:goroutine] Http Server closed:", err)
  }
 }()
 return fmt.Sprintf("http://%s/", address), nil
}
```

```
// http_handler handles the requests coming from the Telemetry API.
// Everytime Telemetry API sends log events, this function will read them from the
 response body
// and put into a synchronous queue to be dispatched later.
// Logging or printing besides the error cases below is not recommended if you have
 subscribed to
// receive extension logs. Otherwise, logging here will cause Telemetry API to send new
 logs for
// the printed lines which may create an infinite loop.
func (s *TelemetryApiListener) http_handler(w http.ResponseWriter, r *http.Request) {
 body, err := ioutil.ReadAll(r.Body)
 if err != nil {
  l.Error("[listener:http_handler] Error reading body:", err)
  return
 }

 // Parse and put the log messages into the queue
 var slice []interface{}
 _ = json.Unmarshal(body, &slice)

 for _, el := range slice {
  s.LogEventsQueue.Put(el)
 }

 l.Info("[listener:http_handler] logEvents received:", len(slice), " LogEventsQueue
 length:", s.LogEventsQueue.Len())
 slice = nil
}
```

Spécification d’un protocole de destination

Lorsque vous vous abonnez pour recevoir de la télémétrie à l’aide de l’API de télémétrie, vous pouvez spécifier un protocole de destination en plus de l’URI de destination :

```
{
    "destination": {
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
    }
}
```

Lambda accepte deux protocoles pour recevoir des télémétries :

- HTTP (recommandé) – Lambda envoie la télémétrie à un point de terminaison HTTP local (`http://sandbox.localdomain:${PORT}/${PATH}`) sous la forme d'un tableau d'enregistrements au format JSON. Le paramètre `$PATH` est facultatif. Lambda ne prend en charge que HTTP, pas HTTPS. Lambda fournit des données de télémétrie via des demandes POST.
- TCP – Lambda envoie la télémétrie à un port TCP au format NDJSON (Newline delimited JSON).

> **Note**
>
> Nous vous recommandons vivement d'utiliser HTTP plutôt que TCP. Avec TCP, la plateforme Lambda ne peut pas confirmer la livraison de la télémétrie à la couche d'application. Par conséquent, si votre extension se bloque, vous risquez de perdre la télémétrie. HTTP ne présente pas cette limitation.

Avant de vous abonner pour recevoir la télémétrie, établissez l'écouteur HTTP local ou le port TCP. Au cours de l'installation, notez ce qui suit :

- Lambda n'envoie la télémétrie qu'à des destinations au sein de l'environnement d'exécution.
- Lambda réessaie d'envoyer la télémétrie (avec interruption) en l'absence d'écouteur, ou si la demande POST rencontre une erreur. Si l'écouteur de télémétrie se bloque, il reprend la réception de la télémétrie après que Lambda a redémarré l'environnement d'exécution.
- Lambda réserve le port 9001. Il n'y a pas d'autres restrictions ou recommandations relatives au numéro de port.

Configuration de l'utilisation de la mémoire et de la mise en mémoire tampon

L'utilisation de la mémoire dans un environnement d'exécution augmente de façon linéaire avec le nombre d'abonnés. Les abonnements consomment des ressources de mémoire, car chacun d'eux ouvre un nouveau tampon mémoire pour stocker les données de télémétrie. L'utilisation de la mémoire tampon contribue à la consommation globale de mémoire dans l'environnement d'exécution.

Lorsque vous vous abonnez pour recevoir des données télémétriques via l'API de télémétrie, vous pouvez mettre en mémoire tampon les données télémétriques et les transmettre aux abonnés par

lots. Pour optimiser l'utilisation de la mémoire, vous pouvez spécifier une configuration de mise en mémoire tampon :

```
{
    "buffering": {
        "maxBytes": 256*1024,
        "maxItems": 1000,
        "timeoutMs": 100
    }
}
```

Paramètre	Description	Valeurs par défaut et limites
`maxBytes`	Le volume maximal de télémétrie (en octets) à mettre en mémoire tampon.	Par défaut : 262 144 Minimum : 262 144 Maximum : 1 048 576
`maxItems`	Le nombre maximal d'événements à mettre en mémoire tampon.	Par défaut : 10 000 Minimum : 1 000 Maximum : 10 000.
`timeoutMs`	La durée maximale (en millisecondes) pour la mise en mémoire tampon d'un lot.	Par défaut : 1 000 Minimum : 25 Maximum : 30 000

Lorsque vous configurez la mise en mémoire tampon, tenez compte des points suivants :

- Si l'un des flux d'entrée est fermé, Lambda vide les journaux. Cela peut se produire si, par exemple, l'environnement d'exécution se bloque.
- Chaque abonné peut spécifier une configuration de mise en mémoire tampon différente dans sa demande d'abonnement.
- Lorsque vous déterminez la taille de la mémoire tampon pour la lecture des données, attendez-vous à recevoir des charges utiles aussi importantes que `2 * maxBytes + metadataBytes`,

`maxBytes` étant un composant de votre configuration de mise en mémoire tampon. Pour évaluer la quantité de `metadataBytes` à prendre en compte, consultez les métadonnées suivantes. Lambda adjoint des métadonnées similaires à celles-ci à chaque enregistrement :

```
{
   "time": "2022-08-20T12:31:32.123Z",
   "type": "function",
   "record": "Hello World"
}
```

- Si l'abonné ne peut pas traiter la télémétrie entrante assez rapidement, ou si votre code de fonction génère un volume de journal très élevé, Lambda peut abandonner des enregistrements pour limiter l'utilisation de la mémoire. Lorsque cela se produit, Lambda envoie un événement `platform.logsDropped`.

Envoi d'une demande d'abonnement à l'API de télémétrie

Les extensions Lambda peuvent s'abonner pour recevoir des données de télémétrie en envoyant une demande d'abonnement à l'API de télémétrie. La demande d'abonnement doit contenir des informations sur les types d'événements auxquels vous voulez que l'extension s'abonne. En outre, la demande peut contenir des informations sur la destination de la livraison et une configuration de mise en mémoire tampon.

Avant d'envoyer une demande d'abonnement, vous devez disposer d'un ID d'extension (`Lambda-Extension-Identifier`). Lorsque vous enregistrez votre extension avec l'API d'extensions, vous obtenez un ID d'extension à partir de la réponse API.

L'abonnement a lieu pendant la phase d'initialisation de l'extension. L'exemple suivant montre une demande HTTP pour s'abonner aux trois flux de télémétrie : télémétrie de la plateforme, journaux des fonctions et journaux des extensions.

```
PUT http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry HTTP/1.1
{
   "schemaVersion": "2025-01-29",
   "types": [
        "platform",
        "function",
        "extension"
   ],
   "buffering": {
```

```
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
    },
    "destination": {
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
    }
}
```

Si la demande réussit, l'abonné reçoit une réponse de succès HTTP 200.

```
HTTP/1.1 200 OK
"OK"
```

Messages entrants de l'API de télémétrie

Après s'être abonnée à l'API de télémétrie, une extension commence automatiquement à recevoir la télémétrie de Lambda via des demandes POST. Chaque corps de requête POST contient un tableau d'objets `Event`. Chaque `Event` est structuré selon le schéma suivant :

```
{
   time: String,
   type: String,
   record: Object
}
```

- La propriété `time` définit le moment où la plateforme Lambda a généré l'événement. Cela est différent du moment où l'événement s'est réellement produit. La valeur de la chaîne `time` est un horodatage au format ISO 8601.
- La propriété `type` définit le type d'événement. Le tableau suivant décrit toutes les valeurs possibles.
- La propriété `record` définit un objet JSON qui contient les données de télémétrie. Le schéma de cet objet JSON dépend de la propriété `type`.

> **Commande d'événements avec invocations simultanées**
>
> Pour les instances gérées Lambda, plusieurs invocations de fonctions peuvent s'exécuter simultanément dans le même environnement d'exécution. Dans ce cas, l'ordre

`platform.start` et les `platform.report` événements ne sont pas garantis entre les différentes invocations simultanées. Les extensions doivent gérer les événements issus de plusieurs invocations exécutées en parallèle et ne doivent pas adopter un ordre séquentiel. Pour attribuer correctement des événements à des invocations spécifiques, les extensions doivent utiliser le `requestId` champ présent dans ces événements de plateforme. Chaque appel possède un identifiant de demande unique qui reste le même pour tous les événements associés à cet appel, ce qui permet aux extensions de corréler correctement les événements même lorsqu'ils arrivent dans le désordre.

Le tableau suivant résume tous les types d'objets `Event` et renvoie à la référence du schéma `Event` de l'API de télémétrie pour chaque type d'événement.

Catégorie	Type d'événement	Description	Schéma d'enregistrement des événements
Événement de plateforme	`platform.initStart`	L'initialisation de la fonction a commencé.	Schéma the section called "`platform.initStart`"
Événement de plateforme	`platform.initRuntimeDone`	L'initialisation de la fonction est terminée.	Schéma the section called "`platform.initRuntimeDone`"
Événement de plateforme	`platform.initReport`	Un rapport d'initialisation de la fonction.	Schéma the section called "`platform.initReport`"
Événement de plateforme	`platform.start`	L'invocation de la fonction a commencé.	Schéma the section called "`platform.start`"
Événement de plateforme	`platform.runtimeDone`	L'environnement d'exécution a fini de traiter un événement	Schéma the section called "`platform.runtimeDone`"

Catégorie	Type d'événement	Description	Schéma d'enregistrement des événements
		avec succès ou échec.	
Événement de plateforme	`platform.report`	Un rapport sur l'invocation de la fonction.	Schéma the section called "`platform.report`"
Événement de plateforme	`platform.restoreStart`	La restauration de l'exécution a commencé.	Schéma the section called "`platform.restoreStart`"
Événement de plateforme	`platform.restoreRuntimeDone`	La restauration de l'exécution est terminée.	Schéma the section called "`platform.restoreRuntimeDone`"
Événement de plateforme	`platform.restoreReport`	Le rapport de restauration de d'exécution.	Schéma the section called "`platform.restoreReport`"
Événement de plateforme	`platform.telemetrySubscription`	L'extension s'est abonnée à l'API de télémétrie.	Schéma the section called "`platform.telemetrySubscription`"
Événement de plateforme	`platform.logsDropped`	Les entrées de journal abandonnées par Lambda.	Schéma the section called "`platform.logsDropped`"
Journaux de fonctions	`function`	Une ligne de journal du code de la fonction.	Schéma the section called "`function`"

Catégorie	Type d'événement	Description	Schéma d'enregistrement des événements
Journaux d'extension	`extension`	Une ligne de journal du code de l'extension.	Schéma the section called "`extension`"

Référence de l'API de télémétrie Lambda

Utilisez le point de terminaison de l'API de télémétrie Lambda pour abonner des extensions aux flux de télémétrie. Vous pouvez extraire le point de terminaison d'API de télémétrie à partir de la variable d'environnement `AWS_LAMBDA_RUNTIME_API`. Pour envoyer une demande d'API, ajoutez la version de l'API (`2022-07-01/`) et `telemetry/`. Par exemple :

```
http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry/
```

Pour la définition de la spécification OpenAPI (OAS) de la version des réponses d'abonnement `2025-01-29`, voir ce qui suit :

- HTTP — telemetry-api-http-schema.zip
- TCP — .zip telemetry-api-tcp-schema

Opérations d'API

- S'abonner

S'abonner

Pour s'abonner à un flux de télémétrie, une extension Lambda peut envoyer une demande d'API d'abonnement.

- Chemin – `/telemetry`
- Méthode – PUT
- En-têtes
  - `Content-Type`: `application/json`
- Paramètres du corps de la demande
  - schemaVersion
    - Obligatoire : oui
    - Type : String
    - Valeurs valides : `"2025-01-29"`, `"2022-12-13"` ou `"2022-07-01"`
    - Remarque : les instances gérées Lambda nécessitent. `"2025-01-29"` Cette version est rétrocompatible avec les fonctions Lambda (par défaut).

- destination – Les paramètres de configuration qui définissent la destination de l'événement de télémétrie et le protocole de livraison de l'événement.
  - Obligatoire : oui
  - Type : objet

```
{
    "protocol": "HTTP",
    "URI": "http://sandbox.localdomain:8080"
}
```

  - protocol – Le protocole que Lambda utilise pour envoyer les données de télémétrie.
    - Obligatoire : oui
    - Type : String
    - Valeurs valides : `"HTTP"|"TCP"`
  - URI – L'URI auquel envoyer les données de télémétrie.
    - Obligatoire : oui
    - Type : String
  - Pour de plus amples informations, veuillez consulter the section called "Spécification d'un protocole de destination".
- types – Les types de télémétrie auxquels vous voulez que l'extension s'abonne.
  - Obligatoire : oui
  - Type : tableau de chaînes
  - Valeurs valides : `"platform"|"function"|"extension"`
- buffering – Les paramètres de configuration de la mise en mémoire tampon des événements.
  - Obligatoire : non
  - Type : objet

```
{
   "buffering": {
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
   }
}
```

    - maxItems – Nombre maximum d'événements à mettre en mémoire tampon.

    - Obligatoire : non
    - Type : Integer
    - Par défaut : 1 000
    - Minimum : 1 000
    - Maximum : 10 000.
- maxBytes – Le volume maximal de télémétrie (en octets) à mettre en mémoire tampon.
    - Obligatoire : non
    - Type : Integer
    - Par défaut : 262 144
    - Minimum : 262 144
    - Maximum : 1 048 576
- timeoutMs – Durée maximum (en millisecondes) de mise en mémoire tampon d'un lot.
    - Obligatoire : non
    - Type : Integer
    - Par défaut : 1 000
    - Minimum : 25
    - Maximum : 30 000
- Pour de plus amples informations, veuillez consulter the section called "Configuration de l'utilisation de la mémoire et de la mise en mémoire tampon".

Exemple de demande d'API d'abonnement

```
PUT http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry HTTP/1.1
{
   "schemaVersion": "2025-01-29",
   "types": [
        "platform",
        "function",
        "extension"
   ],
   "buffering": {
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
   },
```

```
   "destination": {
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
   }
}
```

Si la demande d'abonnement aboutit, l'extension reçoit une réponse HTTP 200 :

```
HTTP/1.1 200 OK
"OK"
```

Si la demande d'abonnement échoue, l'extension reçoit une réponse d'erreur. Par exemple :

```
HTTP/1.1 400 OK
{
    "errorType": "ValidationError",
    "errorMessage": "URI port is not provided; types should not be empty"
}
```

Voici quelques codes de réponse supplémentaires que l'extension peut recevoir :

- 200 – Demande effectuée avec succès.
- 202 – Demande acceptée. Réponse à une demande d'abonnement dans un environnement de test local
- 400 – Demande erronée
- 500 – Erreur de service.

Référence du schéma **Event** de l'API de télémétrie Lambda

Utilisez le point de terminaison de l'API de télémétrie Lambda pour abonner des extensions aux flux de télémétrie. Vous pouvez extraire le point de terminaison d'API de télémétrie à partir de la variable d'environnement `AWS_LAMBDA_RUNTIME_API`. Pour envoyer une demande d'API, ajoutez la version de l'API (`2022-07-01/`) et `telemetry/`. Par exemple :

```
http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry/
```

Pour la définition de la spécification OpenAPI (OAS) de la version des réponses d'abonnement `2025-01-29`, voir ce qui suit :

- HTTP — telemetry-api-http-schema.zip
- TCP — .zip telemetry-api-tcp-schema

Le tableau suivant est un résumé de tous les types d'objets `Event` que l'API de télémétrie prend en charge.

Catégorie	Type d'événement	Description	Schéma d'enregistrement des événements
Événement de plateforme	`platform.initStart`	L'initialisation de la fonction a commencé.	Schéma the section called "`platform.initStart`"
Événement de plateforme	`platform.initRuntimeDone`	L'initialisation de la fonction est terminée.	Schéma the section called "`platform.initRuntimeDone`"
Événement de plateforme	`platform.initReport`	Un rapport d'initialisation de la fonction.	Schéma the section called "`platform.initReport`"

Catégorie	Type d'événement	Description	Schéma d'enregistrement des événements
Événement de plateforme	`platform.start`	L'invocation de la fonction a commencé.	Schéma the section called "`platform.start`"
Événement de plateforme	`platform.runtimeDone`	L'environnement d'exécution a fini de traiter un événement avec succès ou échec.	Schéma the section called "`platform.runtimeDone`"
Événement de plateforme	`platform.report`	Un rapport sur l'invocation de la fonction.	Schéma the section called "`platform.report`"
Événement de plateforme	`platform.restoreStart`	La restauration de l'exécution a commencé.	Schéma the section called "`platform.restoreStart`"
Événement de plateforme	`platform.restoreRuntimeDone`	La restauration de l'exécution est terminée.	Schéma the section called "`platform.restoreRuntimeDone`"
Événement de plateforme	`platform.restoreReport`	Le rapport de restauration de d'exécution.	Schéma the section called "`platform.restoreReport`"
Événement de plateforme	`platform.telemetrySubscription`	L'extension s'est abonnée à l'API de télémétrie.	Schéma the section called "`platform.telemetrySubscription`"

Catégorie	Type d'événement	Description	Schéma d'enregistrement des événements
Événement de plateforme	`platform.logsDropped`	Les entrées de journal abandonnées par Lambda.	Schéma the section called "`platform.logsDropped`"
Journaux de fonctions	`function`	Une ligne de journal du code de la fonction.	Schéma the section called "`function`"
Journaux d'extension	`extension`	Une ligne de journal du code de l'extension.	Schéma the section called "`extension`"

Table des matières

- Types d'objets Event de l'API de télémétrie
  - platform.initStart
  - platform.initRuntimeDone
  - platform.initReport
  - platform.start
  - platform.runtimeDone
  - platform.report
  - platform.restoreStart
  - platform.restoreRuntimeDone
  - platform.restoreReport
  - platform.extension
  - platform.telemetrySubscription
  - platform.logsDropped
  - function
  - extension
- Types d'objets partagés

- InitPhase
- InitReportMetrics
- InitType
- ReportMetrics
- RestoreReportMetrics
- RuntimeDoneMetrics
- Span
- Status
- TraceContext
- TracingType

Types d'objets **`Event`** de l'API de télémétrie

Cette section détaille les types d'objets `Event` que l'API de télémétrie Lambda prend en charge. Dans les descriptions d'événements, un point d'interrogation (?) indique que l'attribut peut ne pas être présent dans l'objet.

`platform.initStart`

Un événement `platform.initStart` indique que la phase d'initialisation de la fonction a commencé. Un objet `Event` `platform.initStart` a la forme suivante :

```
Event: Object
- time: String
- type: String = platform.initStart
- record: PlatformInitStart
```

L'objet `PlatformInitStart` possède les attributs suivants :

- functionName – `String`
- functionVersion – `String`
- InitializationType – Objet `the section called "InitType"`
- instanceId? – `String`
- instanceMaxMemory? – `Integer`
- phase – Objet `the section called "InitPhase"`

- runtimeVersion? – `String`
- runtimeVersionArn? – `String`

Voici un exemple de `Event` de type `platform.initStart` :

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.initStart",
    "record": {
        "initializationType": "on-demand",
        "phase": "init",
        "runtimeVersion": "nodejs-14.v3",
        "runtimeVersionArn": "arn",
        "functionName": "myFunction",
        "functionVersion": "$LATEST",
        "instanceId": "82561ce0-53dd-47d1-90e0-c8f5e063e62e",
        "instanceMaxMemory": 256
    }
}
```

`platform.initRuntimeDone`

Un événement `platform.initRuntimeDone` indique que la phase d'initialisation de la fonction est terminée. Un objet `Event` `platform.initRuntimeDone` a la forme suivante :

```
Event: Object
- time: String
- type: String = platform.initRuntimeDone
- record: PlatformInitRuntimeDone
```

L'objet `PlatformInitRuntimeDone` possède les attributs suivants :

- InitializationType – Objet the section called "InitType"
- phase – Objet the section called "InitPhase"
- status – Objet the section called "Status"
- spans? – Liste d'objets the section called "Span"

Voici un exemple de `Event` de type `platform.initRuntimeDone` :

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.initRuntimeDone",
    "record": {
        "initializationType": "on-demand"
        "status": "success",
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-06-02T12:02:33.913Z",
                "durationMs": 70.5
            }
        ]
    }
}
```

`platform.initReport`

Un événement `platform.initReport` contient un rapport global de la phase d'initialisation de la fonction. Un objet `Event` `platform.initReport` a la forme suivante :

```
Event: Object
- time: String
- type: String = platform.initReport
- record: PlatformInitReport
```

L'objet `PlatformInitReport` possède les attributs suivants :

- errorType? – chaîne
- InitializationType – Objet the section called "InitType"
- phase – Objet the section called "InitPhase"
- metrics – Objet the section called "InitReportMetrics"
- spans? – Liste d'objets the section called "Span"
- status – Objet the section called "Status"

Voici un exemple de `Event` de type `platform.initReport` :

```
{
    "time": "2022-10-12T00:01:15.000Z",
```

```
    "type": "platform.initReport",
    "record": {
        "initializationType": "on-demand",
        "status": "success",
        "phase": "init",
        "metrics": {
            "durationMs": 125.33
        },
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-06-02T12:02:33.913Z",
                "durationMs": 90.1
            }
        ]
    }
}
```

`platform.start`

Un événement `platform.start` indique que la phase d'invocation de la fonction a commencé. Un objet `Event` `platform.start` a la forme suivante :

```
Event: Object
- time: String
- type: String = platform.start
- record: PlatformStart
```

L'objet `PlatformStart` possède les attributs suivants :

- requestId – `String`
- version? – `String`
- tracing? – the section called "TraceContext"

Voici un exemple de `Event` de type `platform.start` :

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.start",
    "record": {
        "requestId": "6d68ca91-49c9-448d-89b8-7ca3e6dc66aa",
```

```
        "version": "$LATEST",
        "tracing": {
            "spanId": "54565fb41ac79632",
            "type": "X-Amzn-Trace-Id",
            "value":
 "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
        }
    }
}
```

`platform.runtimeDone`

Un événement `platform.runtimeDone` indique que la phase d'invocation de la fonction est terminée. Un objet `Event` `platform.runtimeDone` a la forme suivante :

> **Instances gérées Lambda**
>
> L'`platform.runtimeDone`événement n'est pas pris en charge pour les instances gérées Lambda. Les extensions exécutées sur les instances gérées ne recevront pas cet événement car les extensions ne peuvent pas s'abonner à l'`INVOKE`événement sur les instances gérées. En raison du modèle d'exécution simultanée dans lequel plusieurs invocations peuvent être traitées simultanément, les extensions ne peuvent pas effectuer de traitement post-appel pour des invocations individuelles comme elles le font traditionnellement sur les fonctions Lambda (par défaut).
>
> Pour les instances gérées, les `responseLatency` intervalles `responseDuration` et qui y sont normalement inclus `platform.runtimeDone` sont plutôt disponibles lors de l'`platform.report`événement. Consultez the section called “`platform.report`” pour plus de détails.

```
Event: Object
- time: String
- type: String = platform.runtimeDone
- record: PlatformRuntimeDone
```

L'objet `PlatformRuntimeDone` possède les attributs suivants :

- errorType? – `String`
- metrics? – Objet the section called “RuntimeDoneMetrics”

- requestId – `String`
- status – Objet the section called “Status”
- spans? – Liste d’objets the section called “Span”
- tracing? – Objet the section called “TraceContext”

Voici un exemple de `Event` de type `platform.runtimeDone` :

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.runtimeDone",
    "record": {
        "requestId": "6d68ca91-49c9-448d-89b8-7ca3e6dc66aa",
        "status": "success",
        "tracing": {
            "spanId": "54565fb41ac79632",
            "type": "X-Amzn-Trace-Id",
            "value":
 "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
        },
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-08-02T12:01:23:521Z",
                "durationMs": 80.0
            }
        ],
        "metrics": {
            "durationMs": 140.0,
            "producedBytes": 16
        }
    }
}
```

`platform.report`

Un événement `platform.report` contient un rapport global de la phase d’invocation de la fonction. Un objet `Event` `platform.report` a la forme suivante :

Instances gérées Lambda

L'`platform.report`événement pour les instances gérées Lambda a des métriques et des durées différentes par rapport aux fonctions Lambda (par défaut). Pour les instances gérées :

- Portée : contient `responseLatency` et `responseDuration` au lieu de. `extensionOverhead` L'`extensionOverhead`intervalle n'est pas disponible car les extensions ne peuvent pas s'abonner à l'`INVOKE`événement sur les instances gérées en raison du modèle d'exécution simultanée.
- Métriques : inclut uniquement`durationMs`. Les mesures suivantes ne sont pas incluses : `billedDurationMs`,`initDurationMs`,`maxMemoryUsedMB`, et`memorySizeMB`. Ces métriques par appel ne sont pas applicables dans l'environnement d'exécution simultanée. Pour les mesures d'utilisation des ressources, utilisez Surveillance des instances gérées par Lambda Lambda Insights.

```
Event: Object
- time: String
- type: String = platform.report
- record: PlatformReport
```

L'objet `PlatformReport` possède les attributs suivants :

- metrics – Objet `the section called "ReportMetrics"`
- requestId – `String`
- spans? – Liste d'objets `the section called "Span"`
- status – Objet `the section called "Status"`
- tracing? – Objet `the section called "TraceContext"`

Voici un exemple de `Event` de type `platform.report` :

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.report",
    "record": {
        "metrics": {
            "billedDurationMs": 694,
```

```
            "durationMs": 693.92,
            "initDurationMs": 397.68,
            "maxMemoryUsedMB": 84,
            "memorySizeMB": 128
        },
        "requestId": "6d68ca91-49c9-448d-89b8-7ca3e6dc66aa",
    }
}
```

`platform.restoreStart`

Un événement `platform.restoreStart` indique qu'un événement de restauration d'environnement de fonction a commencé. Dans un événement de restauration d'environnement, Lambda crée l'environnement à partir d'un instantané mis en cache plutôt que de l'initialiser à partir de zéro. Pour de plus amples informations, veuillez consulter Lambda SnapStart. Un objet `Event` `platform.restoreStart` a la forme suivante :

```
Event: Object
- time: String
- type: String = platform.restoreStart
- record: PlatformRestoreStart
```

L'objet `PlatformRestoreStart` possède les attributs suivants :

- functionName – `String`
- functionVersion – `String`
- instanceId? – `String`
- instanceMaxMemory? – `String`
- runtimeVersion? – `String`
- runtimeVersionArn? – `String`

Voici un exemple de `Event` de type `platform.restoreStart` :

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.restoreStart",
    "record": {
        "runtimeVersion": "nodejs-14.v3",
```

```
        "runtimeVersionArn": "arn",
        "functionName": "myFunction",
        "functionVersion": "$LATEST",
        "instanceId": "82561ce0-53dd-47d1-90e0-c8f5e063e62e",
        "instanceMaxMemory": 256
    }
}
```

`platform.restoreRuntimeDone`

Un événement `platform.restoreRuntimeDone` indique qu'un événement de restauration d'environnement de fonction s'est terminé. Dans un événement de restauration d'environnement, Lambda crée l'environnement à partir d'un instantané mis en cache plutôt que de l'initialiser à partir de zéro. Pour de plus amples informations, veuillez consulter Lambda SnapStart. Un objet `Event` `platform.restoreRuntimeDone` a la forme suivante :

```
Event: Object
- time: String
- type: String = platform.restoreRuntimeDone
- record: PlatformRestoreRuntimeDone
```

L'objet `PlatformRestoreRuntimeDone` possède les attributs suivants :

- errorType? – `String`
- spans? – Liste d'objets `the section called "Span"`
- status – Objet `the section called "Status"`

Voici un exemple de `Event` de type `platform.restoreRuntimeDone` :

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.restoreRuntimeDone",
    "record": {
        "status": "success",
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-08-02T12:01:23:521Z",
                "durationMs": 80.0
            }
```

```
        ]
    }
}
```

`platform.restoreReport`

Un événement `platform.restoreReport` contient un rapport global d'un événement de restauration de fonction. Un objet `Event` `platform.restoreReport` a la forme suivante :

```
Event: Object
- time: String
- type: String = platform.restoreReport
- record: PlatformRestoreReport
```

L'objet `PlatformRestoreReport` possède les attributs suivants :

- errorType? – chaîne
- metrics? – Objet the section called "RestoreReportMetrics"
- spans? – Liste d'objets the section called "Span"
- status – Objet the section called "Status"

Voici un exemple de `Event` de type `platform.restoreReport` :

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.restoreReport",
    "record": {
        "status": "success",
        "metrics": {
            "durationMs": 15.19
        },
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-08-02T12:01:23:521Z",
                "durationMs": 30.0
            }
        ]
    }
}
```

platform.extension

Un événement extension contient des journaux provenant du code d'extension. Un objet Event extension a la forme suivante :

```
Event: Object
- time: String
- type: String = extension
- record: {}
```

L'objet PlatformExtension possède les attributs suivants :

- events – Liste de String
- nom – String
- state – String

Voici un exemple de Event de type platform.extension :

```
{
    "time": "2022-10-12T00:02:15.000Z",
    "type": "platform.extension",
    "record": {
        "events": [ "INVOKE", "SHUTDOWN" ],
        "name": "my-telemetry-extension",
        "state": "Ready"
    }
}
```

platform.telemetrySubscription

Un événement platform.telemetrySubscription contient des informations sur un abonnement d'extension. Un objet Event platform.telemetrySubscription a la forme suivante :

```
Event: Object
- time: String
- type: String = platform.telemetrySubscription
- record: PlatformTelemetrySubscription
```

L'objet PlatformTelemetrySubscription possède les attributs suivants :

- nom – `String`
- state – `String`
- types – Liste de `String`

Voici un exemple de `Event` de type `platform.telemetrySubscription` :

```
{
    "time": "2022-10-12T00:02:35.000Z",
    "type": "platform.telemetrySubscription",
    "record": {
        "name": "my-telemetry-extension",
        "state": "Subscribed",
        "types": [ "platform", "function" ]
    }
}
```

`platform.logsDropped`

Un événement `platform.logsDropped` contient des informations sur les événements abandonnés. Lambda génère un événement `platform.logsDropped` lorsqu'une fonction génère des journaux à un rythme trop élevé pour que Lambda puisse les traiter. Lorsque Lambda ne parvient pas à envoyer des journaux vers CloudWatch ou vers l'extension abonnée à l'API de télémétrie au rythme où la fonction les produit, il supprime les journaux pour empêcher l'exécution de la fonction de ralentir. Un objet `Event` `platform.logsDropped` a la forme suivante :

```
Event: Object
- time: String
- type: String = platform.logsDropped
- record: PlatformLogsDropped
```

L'objet `PlatformLogsDropped` possède les attributs suivants :

- droppedBytes – `Integer`
- droppedRecords – `Integer`
- reason – `String`

Voici un exemple de `Event` de type `platform.logsDropped` :

```
{
    "time": "2022-10-12T00:02:35.000Z",
    "type": "platform.logsDropped",
    "record": {
        "droppedBytes": 12345,
        "droppedRecords": 123,
        "reason": "Some logs were dropped because the downstream consumer is slower
 than the logs production rate"
    }
}
```

`function`

Un événement `function` contient des journaux provenant du code de fonction. Un objet `Event function` a la forme suivante :

```
Event: Object
- time: String
- type: String = function
- record: {}
```

Le format du champ `record` varie selon que les journaux de votre fonction sont formatés en texte brut ou au format JSON. Pour en savoir plus sur les options de configuration du format de journal, consultez the section called “Formats des journaux”

Voici un exemple `Event` de type `function` où le format du journal est en texte brut :

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "function",
    "record": "[INFO] Hello world, I am a function!"
}
```

Voici un exemple `Event` de type `function` où le journal est au format JSON :

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "function",
    "record": {
        "timestamp": "2022-10-12T00:03:50.000Z",
```

```
        "level": "INFO",
        "requestId": "79b4f56e-95b1-4643-9700-2807f4e68189",
        "message": "Hello world, I am a function!"
    }
}
```

> **Note**
>
> Si la version du schéma que vous utilisez est plus ancienne que la version `2022-12-13`, `"record"` est toujours affiché sous forme de chaîne, même lorsque le format de journalisation de votre fonction est configuré au format JSON. Pour les instances gérées Lambda, vous devez utiliser la version du schéma. `2025-01-29`

`extension`

Un événement `extension` contient des journaux provenant du code d'extension. Un objet `Event` `extension` a la forme suivante :

```
Event: Object
- time: String
- type: String = extension
- record: {}
```

Le format du champ `record` varie selon que les journaux de votre fonction sont formatés en texte brut ou au format JSON. Pour en savoir plus sur les options de configuration du format de journal, consultez the section called “Formats des journaux”

Voici un exemple `Event` de type `extension` où le format du journal est en texte brut :

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "extension",
    "record": "[INFO] Hello world, I am an extension!"
}
```

Voici un exemple `Event` de type `extension` où le journal est au format JSON :

```
{
```

```
    "time": "2022-10-12T00:03:50.000Z",
    "type": "extension",
    "record": {
       "timestamp": "2022-10-12T00:03:50.000Z",
       "level": "INFO",
       "requestId": "79b4f56e-95b1-4643-9700-2807f4e68189",
       "message": "Hello world, I am an extension!"
    }
}
```

> **Note**
>
> Si la version du schéma que vous utilisez est plus ancienne que la version `2022-12-13`, `"record"` est toujours affiché sous forme de chaîne, même lorsque le format de journalisation de votre fonction est configuré au format JSON. Pour les instances gérées Lambda, vous devez utiliser la version du schéma. `2025-01-29`

Types d'objets partagés

Cette section détaille les types d'objets partagés que l'API de télémétrie Lambda prend en charge.

`InitPhase`

Une enum de chaînes de caractères qui décrit la phase où l'étape d'initialisation se produit. Dans la plupart des cas, Lambda exécute le code d'initialisation de la fonction pendant la phase `init`. Toutefois, dans certains cas d'erreur, Lambda peut réexécuter le code d'initialisation de la fonction pendant la phase `invoke`. (Cela s'appelle une init supprimée.)

- Type – `String`
- Valeurs valides : `init|invoke|snap-start`

`InitReportMetrics`

Un objet qui contient des métriques sur une phase d'initialisation.

- Type – `Object`

Un objet `InitReportMetrics` a la forme suivante :

```
InitReportMetrics: Object
- durationMs: Double
```

Voici un exemple d'objet `InitReportMetrics` :

```
{
    "durationMs": 247.88
}
```

`InitType`

Une enum de chaînes de caractères qui décrit comment Lambda a initialisé l'environnement.

- Type – `String`
- Valeurs valides – `on-demand`|`provisioned-concurrency`

`ReportMetrics`

Un objet qui contient des métriques sur une phase terminée.

- Type – `Object`

Un objet `ReportMetrics` a la forme suivante :

```
ReportMetrics: Object
- billedDurationMs: Integer
- durationMs: Double
- initDurationMs?: Double
- maxMemoryUsedMB: Integer
- memorySizeMB: Integer
- restoreDurationMs?: Double
```

Voici un exemple d'objet `ReportMetrics` :

```
{
    "billedDurationMs": 694,
    "durationMs": 693.92,
    "initDurationMs": 397.68,
    "maxMemoryUsedMB": 84,
```

```
    "memorySizeMB": 128
}
```

`RestoreReportMetrics`

Un objet qui contient des métriques sur une phase de restauration terminée.

- Type – `Object`

Un objet `RestoreReportMetrics` a la forme suivante :

```
RestoreReportMetrics: Object
- durationMs: Double
```

Voici un exemple d'objet `RestoreReportMetrics` :

```
{
    "durationMs": 15.19
}
```

`RuntimeDoneMetrics`

Un objet qui contient des métriques sur une phase d'invocation.

- Type – `Object`

Un objet `RuntimeDoneMetrics` a la forme suivante :

```
RuntimeDoneMetrics: Object
- durationMs: Double
- producedBytes?: Integer
```

Voici un exemple d'objet `RuntimeDoneMetrics` :

```
{
    "durationMs": 200.0,
    "producedBytes": 15
}
```

Span

Un objet qui contient des détails sur un span. Un span représente une unité de travail ou une opération dans une trace. Pour plus d'informations sur les spans, consultez Span sur la page de l'API de suivi du site Web de OpenTelemetry Docs.

Lambda prend en charge les span suivants pour l'événement `platform.RuntimeDone` :

- Le span `responseLatency` décrit le temps qu'il a fallu à votre fonction Lambda pour commencer à envoyer la réponse.
- Le span `responseDuration` décrit le temps qu'il a fallu à votre fonction Lambda pour finir d'envoyer la réponse entière.
- Le span `runtimeOverhead` décrit le temps qu'il a fallu à l'environnement d'exécution Lambda pour signaler qu'il était prêt à traiter la prochaine invocation de fonction. C'est le temps qu'il a fallu à l'environnement d'exécution pour appeler l'API d'invocation suivante pour obtenir le prochain événement après avoir renvoyé la réponse de votre fonction.

Voici un exemple d'objet span `responseLatency` :

```
{
        "name": "responseLatency",
        "start": "2022-08-02T12:01:23.521Z",
        "durationMs": 23.02
      }
```

Status

Un objet qui décrit le statut d'une phase d'initialisation ou d'invocation. Si le statut est `failure` ou `error`, l'objet `Status` contient également un champ `errorType` décrivant l'erreur.

- Type – `Object`
- Valeurs de status valides : `success|failure|error|timeout`

TraceContext

Un objet qui décrit les propriétés d'une trace.

- Type – `Object`

Un objet `TraceContext` a la forme suivante :

```
TraceContext: Object
- spanId?: String
- type: TracingType enum
- value: String
```

Voici un exemple d'objet `TraceContext` :

```
{
    "spanId": "073a49012f3c312e",
    "type": "X-Amzn-Trace-Id",
    "value":
 "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
}
```

`TracingType`

Un enum de chaînes de caractères qui décrit le type de trace dans un objet the section called "`TraceContext`".

- Type – `String`
- Valeurs valides – `X-Amzn-Trace-Id`

Conversion des objets **`Event`** de l'API de télémétrie Lambda en OpenTelemetry Spans

Le schéma de l'API de télémétrie AWS Lambda est sémantiquement compatible avec OpenTelemetry (OTel). Cela signifie que vous pouvez convertir vos objets AWS Lambda API de télémétrie `Event` en Spans OpenTelemetry (OTel). Lors de la conversion, vous ne devez pas mapper un seul objet `Event` à un seul OTel Span. Au lieu de cela, vous devez présenter les trois événements liés à une phase du cycle de vie dans un seul OTel Span. Par exemple, les événements `start`, `runtimeDone` et `runtimeReport` représentent un seul appel de fonction. Présentez ces trois événements sous la forme d'un seul OTel Span.

Vous pouvez convertir vos événements en utilisant des événements Span ou des Spans enfants (imbriqués). Les tableaux de cette page décrivent les mappages entre les propriétés du schéma de l'API de télémétrie et les propriétés des OTel Spans pour les deux approches. Pour plus d'informations sur les OTel Spans, consultez Span sur la page de l'API de traçage du site Web de la documentation OpenTelemetry.

Sections

- Mappage vers les OTel Spans avec les événements Span
- Mappage vers des OTel Spans avec des Spans enfants

Mappage vers les OTel Spans avec les événements Span

Dans les tableaux suivants, e représente l'événement provenant de la source de télémétrie.

Mappage des événements *Start

OpenTelemetry	Schéma de l'API de télémétrie Lambda
`Span.Name`	Votre extension génère cette valeur sur la base du champ `type`.
`Span.StartTime`	Utilisez `e.time`.
`Span.EndTime`	S/O, car l'événement n'est pas encore terminé.
`Span.Kind`	Définissez sur `Server`.

OpenTelemetry	Schéma de l'API de télémétrie Lambda
`Span.Status`	Définissez sur `Unset`.
`Span.TraceId`	Analysez l'en-tête AWS X-Ray trouvé dans `e.tracing.value` , puis utilisez la valeur `TraceId`.
`Span.ParentId`	Analysez l'en-tête X-Ray trouvé dans `e.tracing.value` , puis utilisez la valeur `Parent`.
`Span.SpanId`	Utilisez `e.tracing.spanId` si disponible. Sinon, générer un nouveau `SpanId`.
`Span.SpanContext.TraceState`	S/O pour un contexte de trace X-Ray.
`Span.SpanContext.TraceFlags`	Analysez l'en-tête X-Ray trouvé dans `e.tracing.value` , puis utilisez la valeur `Sampled`.
`Span.Attributes`	Votre extension peut ajouter toute valeur personnalisée ici.

Mappage des événements *RuntimeDone

OpenTelemetry	Schéma de l'API de télémétrie Lambda
`Span.Name`	Votre extension génère la valeur basée sur le champ `type`.
`Span.StartTime`	Utilisez `e.time` à partir de l'événement `*Start` correspondant. Sinon, utilisez `e.time - e.metrics.durationMs` .
`Span.EndTime`	S/O, car l'événement n'est pas encore terminé.

OpenTelemetry	Schéma de l'API de télémétrie Lambda
`Span.Kind`	Définissez sur `Server`.
`Span.Status`	Si `e.status` n'est pas égal à `success`, mettez `Error`. Sinon, définissez sur `Ok`.
`Span.Events[]`	Utilisez `e.spans[]` .
`Span.Events[i].Name`	Utilisez `e.spans[i].name` .
`Span.Events[i].Time`	Utilisez `e.spans[i].start` .
`Span.TraceId`	Analysez l'en-tête AWS X-Ray trouvé dans `e.tracing.value` , puis utilisez la valeur `TraceId`.
`Span.ParentId`	Analysez l'en-tête X-Ray trouvé dans `e.tracing.value` , puis utilisez la valeur `Parent`.
`Span.SpanId`	Utilisez le même `SpanId` de l'événement `*Start`. S'il n'est pas disponible, utilisez alors `e.tracing.spanId` , ou générez un nouveau `SpanId`.
`Span.SpanContext.TraceState`	S/O pour un contexte de trace X-Ray.
`Span.SpanContext.TraceFlags`	Analysez l'en-tête X-Ray trouvé dans `e.tracing.value` , puis utilisez la valeur `Sampled`.
`Span.Attributes`	Votre extension peut ajouter toute valeur personnalisée ici.

Mappage des événements *Report

OpenTelemetry	Schéma de l'API de télémétrie Lambda
`Span.Name`	Votre extension génère la valeur basée sur le champ `type`.
`Span.StartTime`	Utilisez `e.time` à partir de l'événement `*Start` correspondant. Sinon, utilisez `e.time - e.metrics.durationMs` .
`Span.EndTime`	Utilisez `e.time`.
`Span.Kind`	Définissez sur `Server`.
`Span.Status`	Utilisez la même valeur que l'événement `*RuntimeDone` .
`Span.TraceId`	Analysez l'en-tête AWS X-Ray trouvé dans `e.tracing.value` , puis utilisez la valeur `TraceId`.
`Span.ParentId`	Analysez l'en-tête X-Ray trouvé dans `e.tracing.value` , puis utilisez la valeur `Parent`.
`Span.SpanId`	Utilisez le même `SpanId` de l'événement `*Start`. S'il n'est pas disponible, utilisez alors `e.tracing.spanId` , ou générez un nouveau `SpanId`.
`Span.SpanContext.TraceState`	S/O pour un contexte de trace X-Ray.
`Span.SpanContext.TraceFlags`	Analysez l'en-tête X-Ray trouvé dans `e.tracing.value` , puis utilisez la valeur `Sampled`.
`Span.Attributes`	Votre extension peut ajouter toute valeur personnalisée ici.

Mappage vers des OTel Spans avec des Spans enfants

Le tableau suivant décrit comment convertir les événements de l'API de télémétrie Lambda en OTel Spans avec des Spans enfants (imbriqués) pour les Spans `*RuntimeDone`. Pour les mappages `*Start` et `*Report`, reportez-vous aux tableaux de the section called "Mappage vers les OTel Spans avec les événements Span", car ils sont les mêmes pour les Spans enfant. Dans cette table, e représente l'événement provenant de la source de télémétrie.

Mappage des événements *RuntimeDone

OpenTelemetry	Schéma de l'API de télémétrie Lambda
`Span.Name`	Votre extension génère la valeur basée sur le champ `type`.
`Span.StartTime`	Utilisez `e.time` à partir de l'événement `*Start` correspondant. Sinon, utilisez `e.time - e.metrics.durationMs` .
`Span.EndTime`	S/O, car l'événement n'est pas encore terminé.
`Span.Kind`	Définissez sur `Server`.
`Span.Status`	Si `e.status` n'est pas égal à `success`, mettez `Error`. Sinon, définissez sur `Ok`.
`Span.TraceId`	Analysez l'en-tête AWS X-Ray trouvé dans `e.tracing.value` , puis utilisez la valeur `TraceId`.
`Span.ParentId`	Analysez l'en-tête X-Ray trouvé dans `e.tracing.value` , puis utilisez la valeur `Parent`.
`Span.SpanId`	Utilisez le même `SpanId` de l'événement `*Start`. S'il n'est pas disponible, utilisez

OpenTelemetry	Schéma de l'API de télémétrie Lambda
	alors `e.tracing.spanId` , ou générez un nouveau `SpanId`.
`Span.SpanContext.TraceState`	S/O pour un contexte de trace X-Ray.
`Span.SpanContext.TraceFlags`	Analysez l'en-tête X-Ray trouvé dans `e.tracing.value` , puis utilisez la valeur `Sampled`.
`Span.Attributes`	Votre extension peut ajouter toute valeur personnalisée ici.
`ChildSpan[i].Name`	Utilisez `e.spans[i].name` .
`ChildSpan[i].StartTime`	Utilisez `e.spans[i].start` .
`ChildSpan[i].EndTime`	Utilisez `e.spans[i].start + e.spans[i].durations` .
`ChildSpan[i].Kind`	Identique au parent `Span.Kind` .
`ChildSpan[i].Status`	Identique au parent `Span.Status` .
`ChildSpan[i].TraceId`	Identique au parent `Span.TraceId` .
`ChildSpan[i].ParentId`	Utilisez le parent `Span.SpanId` .
`ChildSpan[i].SpanId`	Générer un nouveau `SpanId`.
`ChildSpan[i].SpanContext.TraceState`	S/O pour un contexte de trace X-Ray.
`ChildSpan[i].SpanContext.TraceFlags`	Identique au parent `Span.SpanContext.TraceFlags` .

Utilisation de l’API Logs Lambda

Important

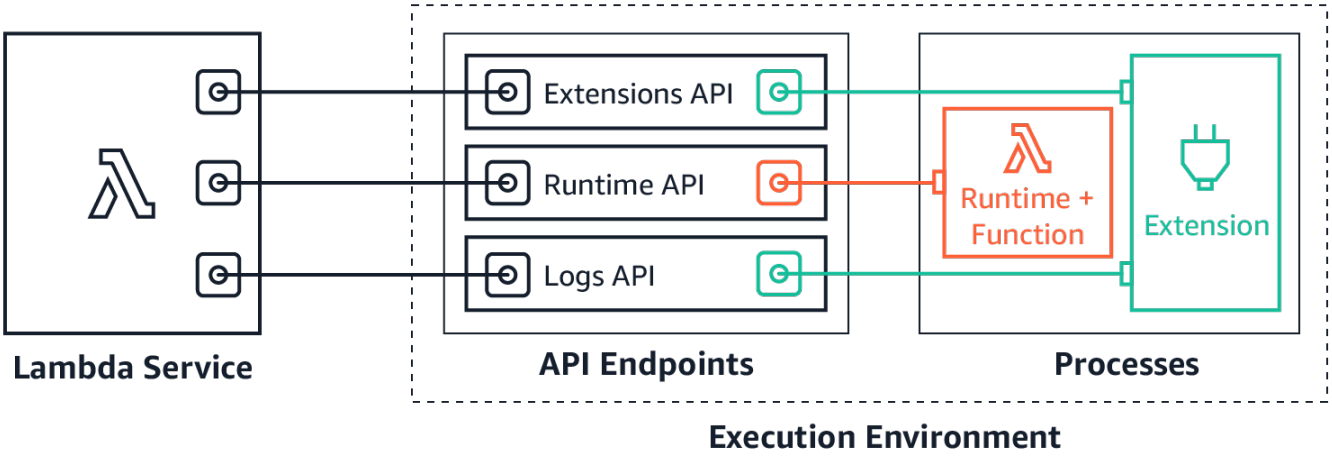
L’API de télémétrie Lambda remplace l’API de journaux Lambda. Bien que l’API de journaux reste entièrement fonctionnelle, nous vous recommandons d’utiliser uniquement l’API de télémétrie à l’avenir. Vous pouvez abonner votre extension à un flux de télémétrie en utilisant l’API de télémétrie ou l’API de journaux. Après vous être abonné à l'aide de l'une de ces API APIs, toute tentative de souscription à l'aide de l'autre API renvoie une erreur.

Les instances gérées Lambda ne prennent pas en charge l'API Logs

Les instances gérées Lambda ne prennent pas en charge l'API Logs. Si vous utilisez des fonctions d'instance gérées, utilisez plutôt l'API de télémétrie. L'API de télémétrie fournit des fonctionnalités améliorées pour collecter et traiter les données de télémétrie à partir de vos fonctions Lambda.

Lambda capture automatiquement les journaux d'exécution et les diffuse vers Amazon. CloudWatch Ce flux de journaux contient les journaux que génèrent votre code de fonction et vos extensions, ainsi que les journaux que Lambda génère dans le cadre de l’appel de fonction.

Les extensions Lambda peuvent utiliser l’API Lambda Runtime Logs pour s’abonner à des flux de journaux directement à partir de l’environnement d’exécution Lambda. Lambda diffuse les journaux vers l’extension qui peut ensuite les traiter, les filtrer et les envoyer à n’importe quelle destination de prédilection.



L'API Logs permet aux extensions de s'abonner à trois flux de journaux différents :

- Journaux de fonction que la fonction Lambda génère et écrit dans `stdout` ou `stderr`.
- Journaux d'extension que le code d'extension génère.
- Journaux de plateforme Lambda qui enregistrent les événements et les erreurs liés aux appels et aux extensions.

> **Note**
>
> Lambda envoie tous les journaux à CloudWatch, même lorsqu'une extension s'abonne à un ou plusieurs flux de journaux.

Rubriques

- S'abonner pour recevoir des journaux
- Utilisation de la mémoire
- Protocoles de destination
- Configuration de mise en mémoire tampon
- Exemple d'abonnement
- Exemple de code pour l'API Logs
- Référence d'API Logs
- Messages de journaux

S'abonner pour recevoir des journaux

Une extension Lambda peut s'abonner aux journaux en envoyant une demande d'abonnement à l'API Logs.

Pour vous abonner afin de recevoir des journaux, vous avez besoin de l'identifiant d'extension (`Lambda-Extension-Identifier`). Enregistrez d'abord l'extension pour recevoir l'identifiant de l'extension. Abonnez-vous ensuite à l'API Logs lors de l'initialisation. Une fois la phase d'initialisation terminée, Lambda ne traite pas les demandes d'abonnement.

> **Note**
>
> L'abonnement à l'API Logs est idempotent. Les demandes d'abonnement en double n'entraînent pas de doublons d'abonnements.

Utilisation de la mémoire

L'utilisation de la mémoire augmente de façon linéaire à mesure que le nombre d'abonnés augmente. Les abonnements consomment des ressources de mémoire car chaque abonnement ouvre un nouveau tampon mémoire pour stocker les journaux. Pour optimiser l'utilisation de la mémoire tampon, vous pouvez ajuster la configuration de la mise en mémoire tampon. L'utilisation de la mémoire tampon compte pour la consommation globale de mémoire dans l'environnement d'exécution.

Protocoles de destination

Vous pouvez choisir l'un des protocoles suivants pour recevoir les journaux :

1. HTTP (recommandé) – Lambda envoie les journaux à un point de terminaison HTTP local (`http://sandbox.localdomain:${PORT}/${PATH}`) sous la forme d'un tableau d'enregistrements au format JSON. Le paramètre `$PATH` est facultatif. Notez que seul HTTP est pris en charge, pas HTTPS. Vous pouvez choisir de recevoir des journaux via PUT ou POST.
2. TCP – Lambda envoie les journaux à un port TCP au format NDJSON (Newline delimited JSON).

Nous vous recommandons d'utiliser HTTP plutôt que TCP. Avec TCP, la plateforme Lambda ne peut pas confirmer la livraison des journaux à la couche d'application. Vous risquez par conséquent de perdre des journaux si votre extension se bloque. HTTP ne partage pas cette limitation.

Nous vous recommandons également de configurer l'écouteur HTTP local ou le port TCP avant de vous abonner pour recevoir les journaux. Au cours de l'installation, notez ce qui suit :

- Lambda n'envoie de journaux qu'à des destinations au sein de l'environnement d'exécution.
- Lambda réessaie d'envoyer les journaux (avec une interruption) s'il n'y a pas d'écouteur, ou si la requête POST ou PUT génère une erreur. Si l'abonné aux journaux se bloque, il continue de recevoir des journaux après que Lambda a redémarré l'environnement d'exécution.
- Lambda réserve le port 9001. Il n'y a pas d'autres restrictions ou recommandations relatives au numéro de port.

Configuration de mise en mémoire tampon

Lambda peut mettre les journaux en mémoire tampon avant de les livrer à l'abonné. Vous pouvez configurer ce comportement dans la demande d'abonnement en spécifiant les champs facultatifs suivants. Notez que Lambda utilise la valeur par défaut pour tout champ que vous ne spécifiez pas.

- timeoutMs – Durée maximum (en millisecondes) de mise en mémoire tampon d'un lot. Par défaut : 1 000. Minimum : 25. Maximum : 30 000.
- maxBytes – Taille maximum (en octets) des journaux à mettre en mémoire tampon. Par défaut : 262 144. Minimum : 262 144. Maximum : 1 048 576.
- maxItems – Nombre maximum d'événements à mettre en mémoire tampon. Par défaut : 10 000. Minimum : 1 000. Maximum : 10 000.

Lors de la configuration de la mise en mémoire tampon, notez les points suivants :

- Lambda vide les journaux en cas de fermeture de l'un des flux d'entrée, par exemple, si le runtime se bloque.
- Chaque abonné peut spécifier une configuration de mise en mémoire tampon différente dans sa demande d'abonnement.
- Tenez compte de la taille de tampon dont vous avez besoin pour lire les données. Attendez-vous à recevoir des charges utiles aussi volumineuses que `2*maxBytes+metadata`, où `maxBytes` est configuré dans la demande d'abonnement. Par exemple, Lambda ajoute les octets de métadonnées suivants à chaque enregistrement :

```
{
"time": "2020-08-20T12:31:32.123Z",
```

```
"type": "function",
"record": "Hello World"
}
```

- Si l'abonné ne peut pas traiter les journaux entrants assez rapidement, Lambda peut supprimer les journaux pour limiter l'utilisation de la mémoire. Pour indiquer le nombre d'enregistrements supprimés, Lambda envoie un journal `platform.logsDropped`. Pour de plus amples informations, veuillez consulter the section called "Lambda : certains journaux de ma fonction n'apparaissent pas".

Exemple d'abonnement

L'exemple suivant montre une demande d'abonnement aux journaux de la plateforme et des fonctions.

```
PUT http://${AWS_LAMBDA_RUNTIME_API}/2020-08-15/logs HTTP/1.1
{ "schemaVersion": "2020-08-15",
  "types": [
      "platform",
      "function"
    ],
  "buffering": {
      "maxItems": 1000,
      "maxBytes": 262144,
      "timeoutMs": 100
    },
  "destination": {
    "protocol": "HTTP",
    "URI": "http://sandbox.localdomain:8080/lambda_logs"
  }
}
```

Si la demande réussit, l'abonné reçoit une réponse de succès HTTP 200.

```
HTTP/1.1 200 OK
"OK"
```

Exemple de code pour l'API Logs

Pour un exemple de code montrant comment envoyer des journaux vers une destination personnalisée, consultez la section Utilisation d' AWS Lambda extensions pour envoyer des journaux vers des destinations personnalisées sur le blog AWS Compute.

Pour des exemples de code Python et Go montrant comment développer une extension Lambda de base et s'abonner à l'API Logs, voir AWS Lambda Extensions dans le référentiel AWS Samples GitHub . Pour plus d'informations sur la génération d'une extension Lambda, consultez the section called "API d'extensions".

Référence d'API Logs

Vous pouvez extraire le point de terminaison d'API Logs à partir de la variable d'environnement `AWS_LAMBDA_RUNTIME_API`. Pour envoyer une demande d'API, utilisez le préfixe `2020-08-15/` avant le chemin d'accès de l'API. Par exemple :

```
http://${AWS_LAMBDA_RUNTIME_API}/2020-08-15/logs
```

La spécification OpenAPI pour la version 2020-08-15 de l'API Logs est disponible ici : .zip logs-api-request

S'abonner

Pour s'abonner à un ou plusieurs des flux de journal disponibles dans l'environnement d'exécution Lambda, les extensions envoient une demande d'API Subscribe.

Chemin – `/logs`

Méthode – PUT

Paramètres de corps

`destination` – Voir the section called "Protocoles de destination". Obligatoire : oui. Type : chaînes.

`buffering` – Voir the section called "Configuration de mise en mémoire tampon". Requis : non. Type : chaînes.

`types` – Tableau des types de journaux à recevoir. Obligatoire : oui. Type : tableau de chaînes. Valeurs valides : « plateforme », « fonction», « extension ».

`schemaVersion` – Obligatoire : non. Valeur par défaut : « 2020-08-15 ». Attribuez la valeur « 2021-03-18 » pour que l'extension reçoive les messages `platform.runtimeDone`.

Paramètres de réponse

Les spécifications OpenAPI pour les réponses d'abonnement version 2020-08-15 sont disponibles pour les protocoles HTTP et TCP :

- HTTP : logs-api-http-response.zip
- TCP : .zip logs-api-tcp-response

Codes de réponse

- 200 – Demande effectuée avec succès.
- 202 – Demande acceptée. Réponse à une demande d'abonnement pendant les tests locaux.
- 4XX – Demande erronée.
- 500 – Erreur de service.

Si la demande réussit, l'abonné reçoit une réponse de succès HTTP 200.

```
HTTP/1.1 200 OK
"OK"
```

Si la demande échoue, l'abonné reçoit une réponse d'erreur. Exemples :

```
HTTP/1.1 400 OK
{
    "errorType": "Logs.ValidationError",
    "errorMessage": URI port is not provided; types should not be empty"
}
```

Messages de journaux

L'API Logs permet aux extensions de s'abonner à trois flux de journaux différents :

- Fonction – Journaux que la fonction Lambda génère et écrit dans `stdout` ou `stderr`.
- Extension – Journaux que le code d'extension génère.

- Plateforme – Journaux que la plateforme de runtime génère, qui consignent des événements et des erreurs liés aux appels et aux extensions.

Rubriques

- Journaux de fonctions
- Journaux d'extension
- Journaux de plateforme

Journaux de fonctions

La fonction Lambda et les extensions internes génèrent des journaux de fonction et les écrivent dans `stdout` ou `stderr`.

L'exemple suivant montre le format d'un message de journal de fonction. { "time": "2020-08-20T12:31:32.123Z", "type": "function", "record": "ERROR encountered. Stack trace:\n\my-function (line 10)\n" }

Journaux d'extension

Les extensions peuvent générer des journaux d'extensions. Le format du journal est le même que pour un journal de fonction.

Journaux de plateforme

Lambda génère des messages de journal pour des événements de plateforme tels que `platform.start`, `platform.end` et `platform.fault`.

Vous pouvez éventuellement vous abonner à la version 2021-03-18 du schéma de l'API Logs, qui inclut le message de journal `platform.runtimeDone`.

Exemple de messages du journal de la plateforme

L'exemple suivant montre les journaux de début et de fin de la plateforme. Ces journaux indiquent l'heure de début et de fin de l'appel pour l'appel spécifié par le requestId.

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.start",
```

```
    "record": {"requestId": "6f7f0961f83442118a7af6fe80b88d56"}
}
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.end",
    "record": {"requestId": "6f7f0961f83442118a7af6fe80b88d56"}
}
```

La plateforme. initRuntimeDoneun message de journal indique l'état de la `Runtime init` sous-phase, qui fait partie de la phase du cycle de vie d'initialisation. Lorsque `Runtime init` est réussie, l'environnement d'exécution envoie une demande d'API d'exécution `/next` (pour les types d'initialisation `on-demand` et `provisioned-concurrency`) ou `restore/next` (pour le type d'initialisation `snap-start`). L'exemple suivant montre une plateforme performante. initRuntimeDonemessage de journal pour le type `snap-start` d'initialisation.

```
{
  "time":"2022-07-17T18:41:57.083Z",
  "type":"platform.initRuntimeDone",
  "record":{
      "initializationType":"snap-start",
      "status":"success"
  }
}
```

Le message de journal platform.initReport indique la durée de la phase `Init` et le nombre de millisecondes qui vous ont été facturées pendant cette phase. Lorsque le type d'initialisation est `provisioned-concurrency`, Lambda envoie ce message pendant l'appel. Lorsque le type d'initialisation est `snap-start`, Lambda envoie ce message après avoir restauré l'instantané. L'exemple suivant montre un message de journal platform.initReport pour le type d'initialisation `snap-start`.

```
{
  "time":"2022-07-17T18:41:57.083Z",
  "type":"platform.initReport",
  "record":{
      "initializationType":"snap-start",
      "metrics":{
          "durationMs":731.79,
          "billedDurationMs":732
          }
  }
```

```
}
```

Le journal du rapport de la plateforme inclut des métriques relatives à l'appel spécifié par le requestId. Le champ `initDurationMs` n'est inclus dans le journal que si l'appel comprend un démarrage à froid. Si le suivi AWS X-Ray est actif, le journal inclut les métadonnées X-Ray. L'exemple suivant montre un journal de rapport de la plateforme pour un appel qui comprend un démarrage à froid.

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.report",
    "record": {"requestId": "6f7f0961f83442118a7af6fe80b88d56",
        "metrics": {"durationMs": 101.51,
            "billedDurationMs": 300,
            "memorySizeMB": 512,
            "maxMemoryUsedMB": 33,
            "initDurationMs": 116.67
        }
    }
}
```

Le journal des pannes de la plateforme capture les erreurs d'exécution ou d'environnement d'exécution. L'exemple suivant montre un message de journal des erreurs de plateforme.

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.fault",
    "record": "RequestId: d783b35e-a91d-4251-af17-035953428a2c Process exited before
 completing request"
}
```

> **Note**
>
> AWS met actuellement en œuvre des modifications du service Lambda. En raison de ces modifications, vous pouvez constater des différences mineures entre la structure et le contenu des messages du journal système et des segments de suivi émis par les différentes fonctions Lambda de votre Compte AWS.
> L'une des sorties de journal concernées par cette modification est le champ `"record"` du journal des erreurs de plateforme. Les exemples suivants montrent des champs `"record"` illustratifs dans les anciens et les nouveaux formats. Le nouveau format de journal des erreurs contient un message plus concis

Ces modifications seront mises en œuvre au cours des prochaines semaines, et toutes les fonctions, Régions AWS sauf en Chine et dans les GovCloud régions, seront transférées pour utiliser le nouveau format des messages de journal et des segments de trace.

Example enregistrement du journal des erreurs de plateforme (ancien format)

```
"record":"RequestId: ...\tError: Runtime exited with error: exit status
 255\nRuntime.ExitError"
```

Example enregistrement du journal des erreurs de plateforme (nouveau format)

```
"record":"RequestId: ... Status: error\tErrorType: Runtime.ExitError"
```

Lambda génère un journal d'extension de plateforme quand une extension s'enregistre auprès de l'API d'extensions. L'exemple suivant montre un message d'extension de la plateforme.

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.extension",
    "record": {"name": "Foo.bar",
        "state": "Ready",
        "events": ["INVOKE", "SHUTDOWN"]
     }
}
```

Lambda génère un journal d'abonnement aux journaux de plateforme quand une extension s'abonne à l'API Logs. L'exemple suivant montre un message d'abonnement aux journaux.

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.logsSubscription",
    "record": {"name": "Foo.bar",
        "state": "Subscribed",
        "types": ["function", "platform"],
    }
}
```

Lambda génère un journal des journaux de plateforme supprimés quand une extension n'est pas en mesure de traiter le nombre de journaux qu'elle reçoit. L'exemple suivant montre un message de journal `platform.logsDropped`.

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.logsDropped",
    "record": {"reason": "Consumer seems to have fallen behind as it has not
 acknowledged receipt of logs.",
        "droppedRecords": 123,
        "droppedBytes" 12345
    }
}
```

Le message de journal platform.restoreStart indique l'heure à laquelle la phase `Restore` a démarré (type d'initialisation `snap-start` uniquement). Exemple :

```
{
  "time":"2022-07-17T18:43:44.782Z",
  "type":"platform.restoreStart",
  "record":{}
}
```

Le message de journal platform.restoreReport indique la durée de la phase `Restore` et le nombre de millisecondes qui vous ont été facturées pendant cette phase (type d'initialisation `snap-start` uniquement). Exemple :

```
{
  "time":"2022-07-17T18:43:45.936Z",
  "type":"platform.restoreReport",
  "record":{
      "metrics":{
          "durationMs":70.87,
          "billedDurationMs":13
      }
  }
}
```

Messages **runtimeDone** de plateforme

Si vous définissez la version du schéma sur « 2021-03-18 » dans la demande d'abonnement, Lambda envoie un message `platform.runtimeDone` une fois l'appel de fonction terminé avec succès ou avec une erreur. L'extension peut utiliser ce message pour arrêter toute la collecte de données télémétriques pour cet appel de fonction.

La spécification OpenAPI pour le type d'événement Log dans la version de schéma 2021-03-18 est disponible ici : schema-2021-03-18.zip

Lambda génère le message de journal `platform.runtimeDone` quand le runtime envoie une demande d'API de runtime `Next` ou `Error`. Le journal `platform.runtimeDone` informe les utilisateurs de l'API Logs que l'appel de fonction se termine. Les extensions peuvent utiliser ces informations pour décider quand envoyer toutes les données télémétriques collectées au cours de cet appel.

Exemples

Lambda envoie le message `platform.runtimeDone` après que le runtime a envoyé la demande NEXT à l'issue de l'appel de fonction. Les exemples suivants présentent des messages pour chacune des valeurs de statut : succès, échec et expiration du délai.

Example Exemple de message de réussite

```
{
    "time": "2021-02-04T20:00:05.123Z",
    "type": "platform.runtimeDone",
    "record": {
       "requestId":"6f7f0961f83442118a7af6fe80b88",
       "status": "success"
    }
}
```

Example Exemple de message d'échec

```
{
   "time": "2021-02-04T20:00:05.123Z",
   "type": "platform.runtimeDone",
   "record": {
      "requestId":"6f7f0961f83442118a7af6fe80b88",
      "status": "failure"
```

```
   }
}
```

Example Exemple de message d'expiration du délai

```
{
   "time": "2021-02-04T20:00:05.123Z",
   "type": "platform.runtimeDone",
   "record": {
      "requestId":"6f7f0961f83442118a7af6fe80b88",
      "status": "timeout"
  }
}
```

Example Exemple de plateforme. restoreRuntimeDone message (type **`snap-start`** d'initialisation uniquement)

La plateforme. restoreRuntimeDoneun message de journal indique si la `Restore` phase a réussi ou non. Lambda envoie ce message lorsque l'environnement d'exécution envoie une demande d'API d'exécution `restore/next`. Il existe trois statuts possibles : succès, échec et dépassement de délai. L'exemple suivant montre une plateforme performante. restoreRuntimeDonemessage de journal.

```
{
  "time":"2022-07-17T18:43:45.936Z",
  "type":"platform.restoreRuntimeDone",
  "record":{
      "status":"success"
  }
}
```

Résolution des problèmes dans Lambda

Les rubriques suivantes fournissent des conseils de dépannage pour les erreurs et les problèmes que vous pouvez rencontrer lors de l'utilisation de l'API, de la console ou des outils Lambda. Si vous rencontrez un problème qui n'est pas répertorié ici, vous pouvez utiliser le bouton Commentaire sur cette page pour le signaler.

Pour plus de conseils de dépannage et de réponses aux questions courantes de support, visitez le Centre de connaissances AWS.

Pour plus d'informations sur le débogage et le dépannage des applications Lambda, consultez Débogage dans Serverless Land.

Rubriques

- Résoudre les problèmes de configuration dans Lambda
- Résoudre les problèmes de déploiement dans Lambda
- Résoudre les problèmes d'invocation dans Lambda
- Résoudre les problèmes d'exécution dans Lambda
- Résoudre les problèmes de mappage des sources d'événements dans Lambda
- Résolution des problèmes de réseaux dans Lambda

Résoudre les problèmes de configuration dans Lambda

Les paramètres de configuration de votre fonction peuvent avoir un impact sur les performances globales et le comportement de votre fonction Lambda. Ils peuvent ne pas provoquer d'erreurs de fonctionnement réelles, mais entraîner des délais et des résultats inattendus.

Les rubriques suivantes fournissent des conseils de dépannage pour les problèmes courants que vous pourriez rencontrer avec les paramètres de configuration des fonctions Lambda.

Rubriques

- Configurations de mémoire
- Configurations dépendant du processeur
- Délais
- Fuite de mémoire entre les invocations

- Résultats asynchrones renvoyés à une invocation ultérieure

Configurations de mémoire

Vous pouvez configurer une fonction Lambda à utiliser entre 128 Mo et 10 240 Mo de mémoire. Par défaut, la plus petite quantité de mémoire est affectée à toutes les fonctions créées dans la console. De nombreuses fonctions Lambda sont performantes à ce réglage le plus bas. Toutefois, si vous importez de grandes bibliothèques de code ou si vous effectuez des tâches gourmandes en mémoire 128 Mo ne suffisent pas.

Si vos fonctions s'exécutent beaucoup plus lentement que prévu, la première étape consiste à augmenter le paramètre de mémoire. Cela vous permet de résoudre le goulet d'étranglement et peut améliorer les performances de votre fonction pour les fonctions dépendant de la mémoire.

Configurations dépendant du processeur

Pour les opérations de calcul intensif, si les performances de votre fonction sont plus lentes que prévu, cela peut être dû au fait que votre fonction est dépendante du processeur. Dans ce cas, la capacité de calcul de la fonction ne peut pas suivre le rythme de la tâche.

Lambda ne vous permet pas de modifier directement la configuration du CPU ; ce dernier est contrôlé indirectement via les paramètres de mémoire. Le service Lambda alloue proportionnellement plus de processeur virtuel à mesure que vous allouez de la mémoire. Avec 1,8 Go de mémoire, un processeur virtuel entier est alloué à une fonction Lambda et, au-dessus de ce niveau, elle a accès à plusieurs cœurs de processeurs virtuels. Avec 10 240 Mo, elle dispose de six processeurs virtuels disponibles. En d'autres termes, vous pouvez améliorer les performances en augmentant l'allocation de mémoire, même si la fonction n'utilise pas toute la mémoire.

Délais

Les délais d'expiration des fonctions Lambda peuvent être définis entre 1 et 900 secondes (15 minutes). Par défaut, la console Lambda définit ce paramètre sur 3 secondes. La valeur du délai d'expiration est une soupape de sécurité qui garantit que les fonctions ne s'exécutent pas indéfiniment. Une fois le délai d'expiration atteint, Lambda arrête l'invocation de fonction.

Si une valeur de délai d'expiration est définie sur une valeur proche de la durée moyenne d'une fonction, cela augmente le risque que la fonction expire de façon inattendue. La durée d'une fonction peut varier en fonction de la quantité de données transférées et traitées, ainsi que de la latence des

services avec lesquels la fonction interagit. Les causes courantes de délai d'expiration comprennent les suivantes :

- Lorsque vous téléchargez des données depuis des compartiments S3 ou d'autres entrepôts de données, le téléchargement est plus important ou prend plus de temps que la moyenne.
- Une fonction envoie une requête à un autre service, qui met plus de temps à répondre.
- Les paramètres fournis à une fonction nécessitent une complexité de calcul accrue dans la fonction, ce qui fait que l'invocation prend plus de temps.

Lors des tests de votre application, veillez à ce que ceux-ci reflètent fidèlement la taille et la quantité des données, et à ce que les valeurs des paramètres soient réalistes. Il est important d'utiliser des jeux de données se situant dans les limites supérieures de ce que l'on peut raisonnablement attendre de votre charge de travail.

En outre, implémentez des limites supérieures à votre charge de travail dans la mesure du possible. Dans cet exemple, l'application peut utiliser une limite de taille maximale pour chaque type de fichier. Vous pouvez ensuite tester les performances de votre application pour une gamme de tailles de fichiers attendues, jusqu'aux limites maximales incluses.

Fuite de mémoire entre les invocations

Les variables globales et les objets stockés dans la phase INIT d'une invocation Lambda conservent leur état entre les invocations à chaud. Ils ne sont complètement réinitialisés que lorsque l'environnement d'exécution est exécuté pour la première fois (également appelé « démarrage à froid »). Toutes les variables stockées dans le gestionnaire sont détruites lorsque le gestionnaire se ferme. Il est recommandé d'utiliser la phase INIT pour configurer les connexions aux bases de données, charger des bibliothèques, créer des caches et charger des actifs immuables.

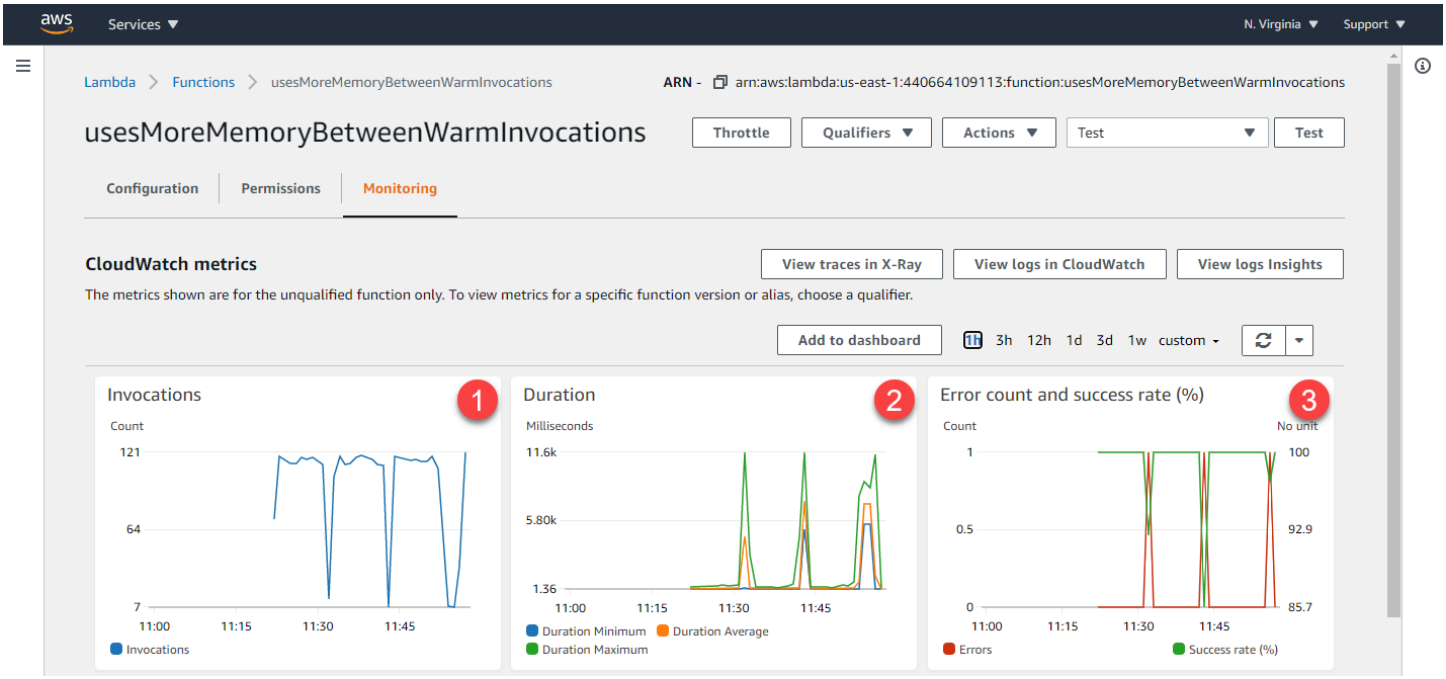
Lorsque vous utilisez des bibliothèques tierces pour plusieurs invocations dans le même environnement d'exécution, consultez leur documentation pour en savoir plus sur leur utilisation dans un environnement de calcul sans serveur. Certaines bibliothèques de connexion à la base de données et de journalisation peuvent enregistrer des résultats d'invocation intermédiaires et d'autres données. Cela entraîne une augmentation de l'utilisation de la mémoire de ces bibliothèques lors des invocations à chaud ultérieures. Si c'est le cas, il se peut que la fonction Lambda soit à court de mémoire, même si votre code personnalisé élimine correctement les variables.

Ce problème affecte les invocations survenant dans des environnements d'exécution à chaud. Par exemple, le code suivant crée une fuite de mémoire entre les invocations. La fonction Lambda

consomme de la mémoire supplémentaire à chaque invocation en augmentant la taille d'un tableau global :

```
let a = []

exports.handler = async (event) => {
    a.push(Array(100000).fill(1))
}
```

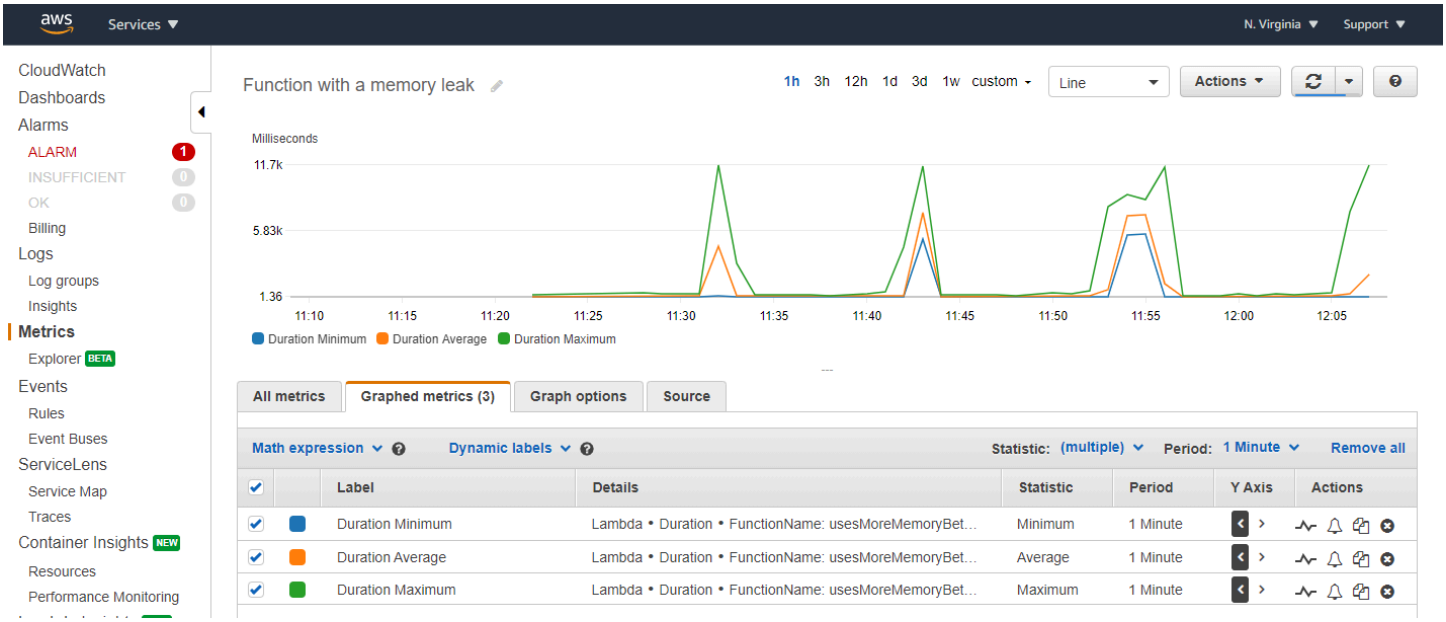
Configuré avec 128 Mo de mémoire, après avoir invoqué cette fonction 1 000 fois, l'onglet Surveillance de la fonction Lambda affiche les modifications typiques des invocations, de la durée et du nombre d'erreurs en cas de fuite de mémoire :



1. Invocations : un taux de transaction stable est interrompu périodiquement, car les invocations prennent plus de temps à être effectuées. À son état stable, la fuite de mémoire ne consomme pas toute la mémoire allouée à la fonction. À mesure que les performances se dégradent, le système d'exploitation appelle le stockage local pour s'adapter à l'augmentation de la mémoire requise par la fonction, ce qui réduit le nombre de transactions effectuées.
2. Durée : avant que la fonction ne soit à court de mémoire, elle termine les invocations au rythme constant d'un nombre de millisecondes à deux chiffres. Au fur et à mesure que les appels se produisent, la durée s'allonge d'un ordre de grandeur.

3. Nombre d'erreurs : lorsque la fuite de mémoire dépasse la mémoire allouée, les erreurs de fonction finissent par se produire en raison du dépassement du délai d'expiration par le calcul, ou l'environnement d'exécution arrête la fonction.

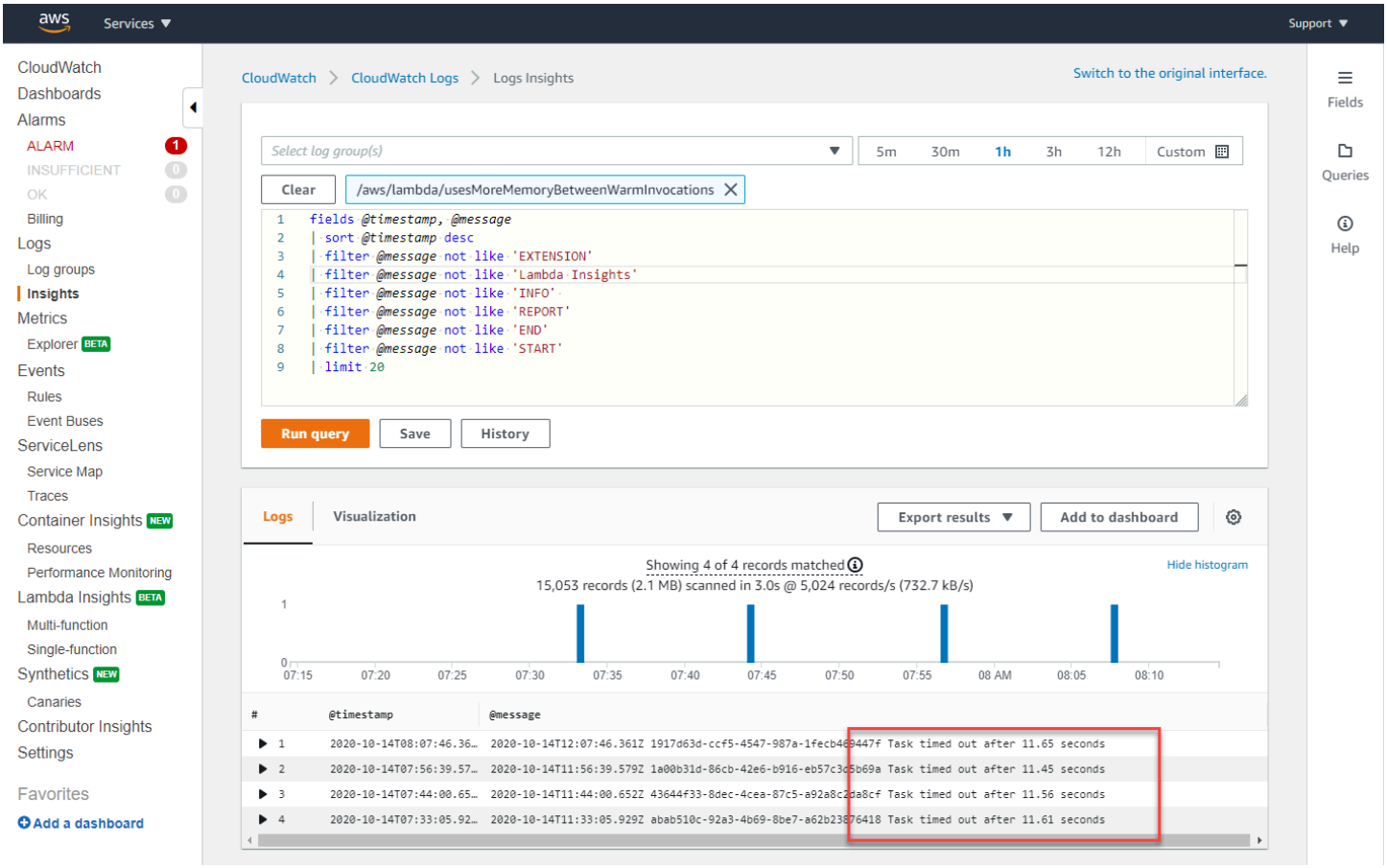
Après l'erreur, Lambda redémarre l'environnement d'exécution, ce qui explique pourquoi on observe un retour à l'état d'origine dans les trois graphiques. L'extension des métriques CloudWatch en fonction de la durée permet d'obtenir plus de détails sur les statistiques de durée minimale, maximale et moyenne :



Pour identifier les erreurs générées lors des 1 000 invocations, vous pouvez utiliser le langage de requête CloudWatch Insights. La requête suivante exclut les journaux d'information pour signaler uniquement les erreurs :

```
fields @timestamp, @message
| sort @timestamp desc
| filter @message not like 'EXTENSION'
| filter @message not like 'Lambda Insights'
| filter @message not like 'INFO'
| filter @message not like 'REPORT'
| filter @message not like 'END'
| filter @message not like 'START'
```

Lorsqu'elle est exécutée sur le groupe de journaux pour cette fonction, cela montre que les délais d'expiration sont à l'origine des erreurs périodiques :



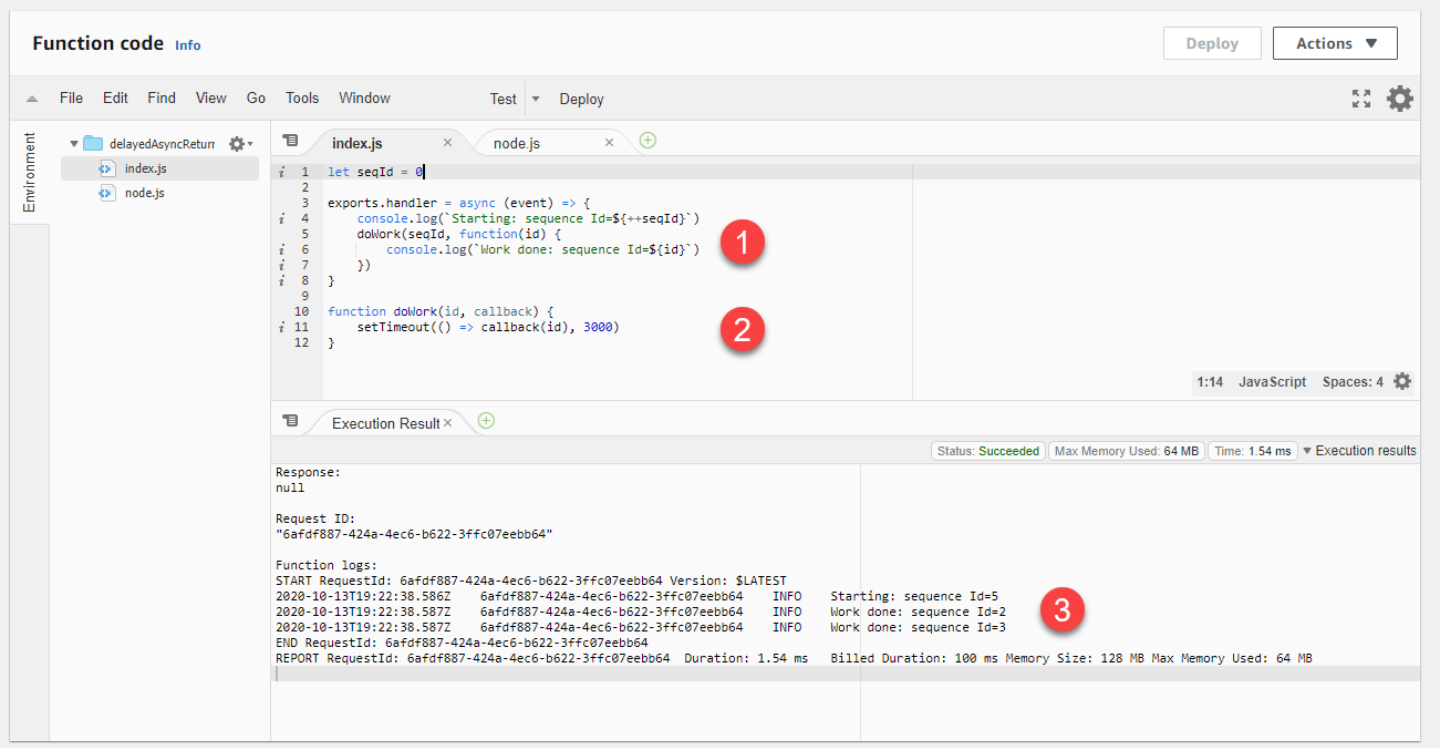
Résultats asynchrones renvoyés à une invocation ultérieure

Pour le code de fonction qui utilise des modèles asynchrones, il est possible que les résultats du rappel d'une invocation soient renvoyés lors d'une future invocation. Cet exemple utilise Node.js, mais la même logique peut s'appliquer à d'autres environnements d'exécution utilisant des modèles asynchrones. La fonction utilise la syntaxe de rappel traditionnelle en JavaScript. Elle appelle une fonction asynchrone avec un compteur incrémentiel qui suit le nombre d'invocations :

```
let seqId = 0

exports.handler = async (event, context) => {
    console.log(`Starting: sequence Id=${++seqId}`)
    doWork(seqId, function(id) {
        console.log(`Work done: sequence Id=${id}`)
    })
}

function doWork(id, callback) {
    setTimeout(() => callback(id), 3000)
```

```
}
```

Lorsqu'ils sont invoqués plusieurs fois de suite, les résultats des rappels apparaissent lors des invocations suivantes :



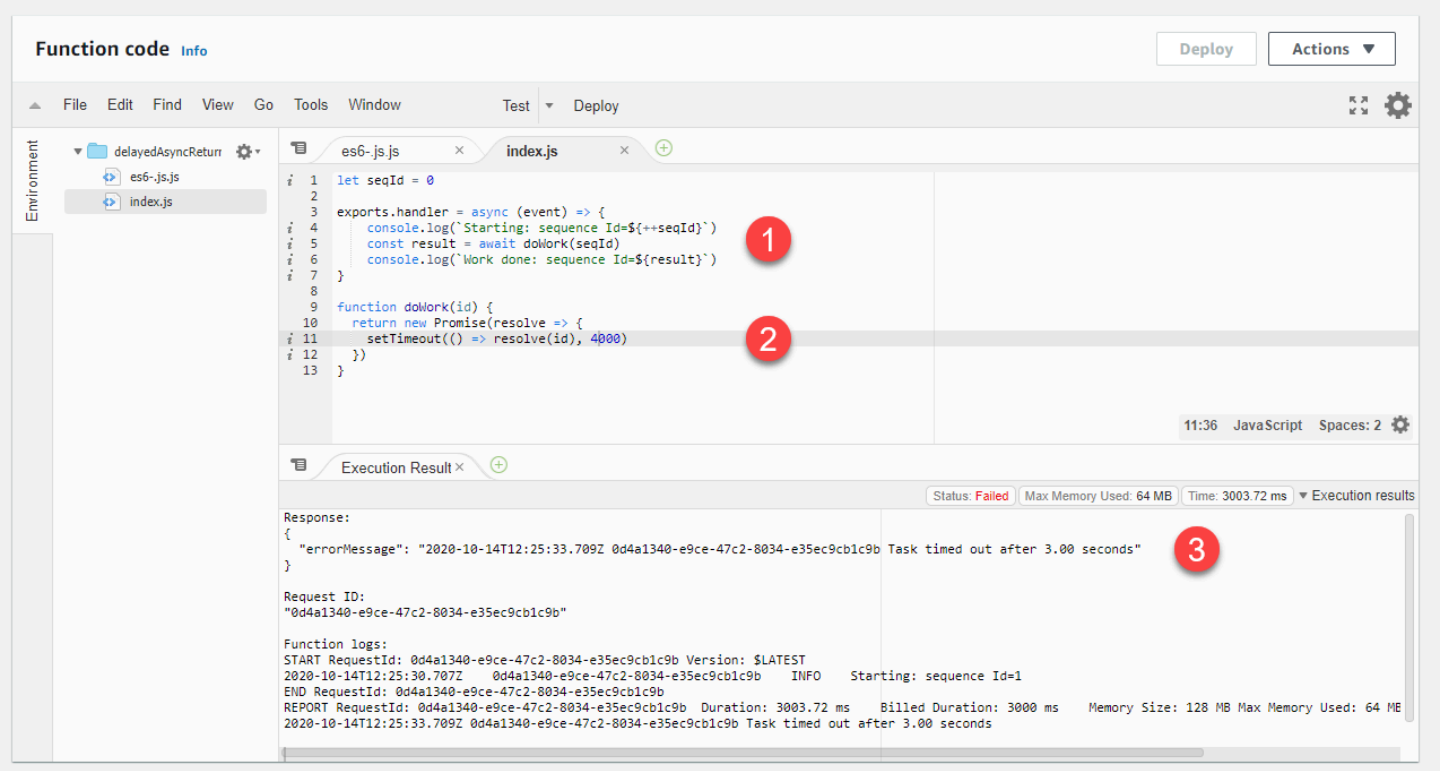
1. Le code appelle la fonction `doWork` en fournissant une fonction de rappel comme dernier paramètre.
2. La fonction `doWork` met un certain temps à s'exécuter avant d'invoquer le rappel.
3. La journalisation de la fonction indique que l'invocation prend fin avant la fin de l'exécution de la fonction `doWork`. De plus, après le démarrage d'une itération, les rappels des itérations précédentes sont traités, comme indiqué dans les journaux.

En JavaScript, les rappels asynchrones sont gérés par une boucle d'événements. D'autres environnements d'exécution utilisent des mécanismes différents pour gérer la simultanéité. Lorsque l'environnement d'exécution de la fonction prend fin, Lambda gèle l'environnement jusqu'à la prochaine invocation. Après sa reprise, JavaScript poursuit le traitement de la boucle d'événements, qui inclut dans ce cas un rappel asynchrone issu d'une invocation précédente. Sans ce contexte, il peut sembler que la fonction exécute du code sans raison et renvoie des données arbitraires. En réalité, il s'agit vraiment d'un artefact de la façon dont la simultanéité d'exécution et les environnements d'exécution interagissent.

Cela crée le risque que les données privées d'une précédente invocation apparaissent lors d'une invocation ultérieure. Il existe deux façons d'empêcher ou de détecter ce comportement. Tout d'abord, JavaScript fournit les mots clés async et await afin de simplifier le développement asynchrone et de forcer l'exécution du code à attendre la fin d'un appel asynchrone. La fonction ci-dessus peut être réécrite en utilisant cette approche comme suit :

```
let seqId = 0
exports.handler = async (event) => {
    console.log(`Starting: sequence Id=${++seqId}`)
    const result = await doWork(seqId)
    console.log(`Work done: sequence Id=${result}`)
}

function doWork(id) {
  return new Promise(resolve => {
    setTimeout(() => resolve(id), 4000)
  })
}
```

L'utilisation de cette syntaxe empêche le gestionnaire de se fermer avant la fin de la fonction asynchrone. Dans ce cas, si le rappel prend plus de temps que le délai d'expiration de la fonction Lambda, la fonction générera une erreur au lieu de renvoyer le résultat du rappel lors d'une invocation ultérieure :

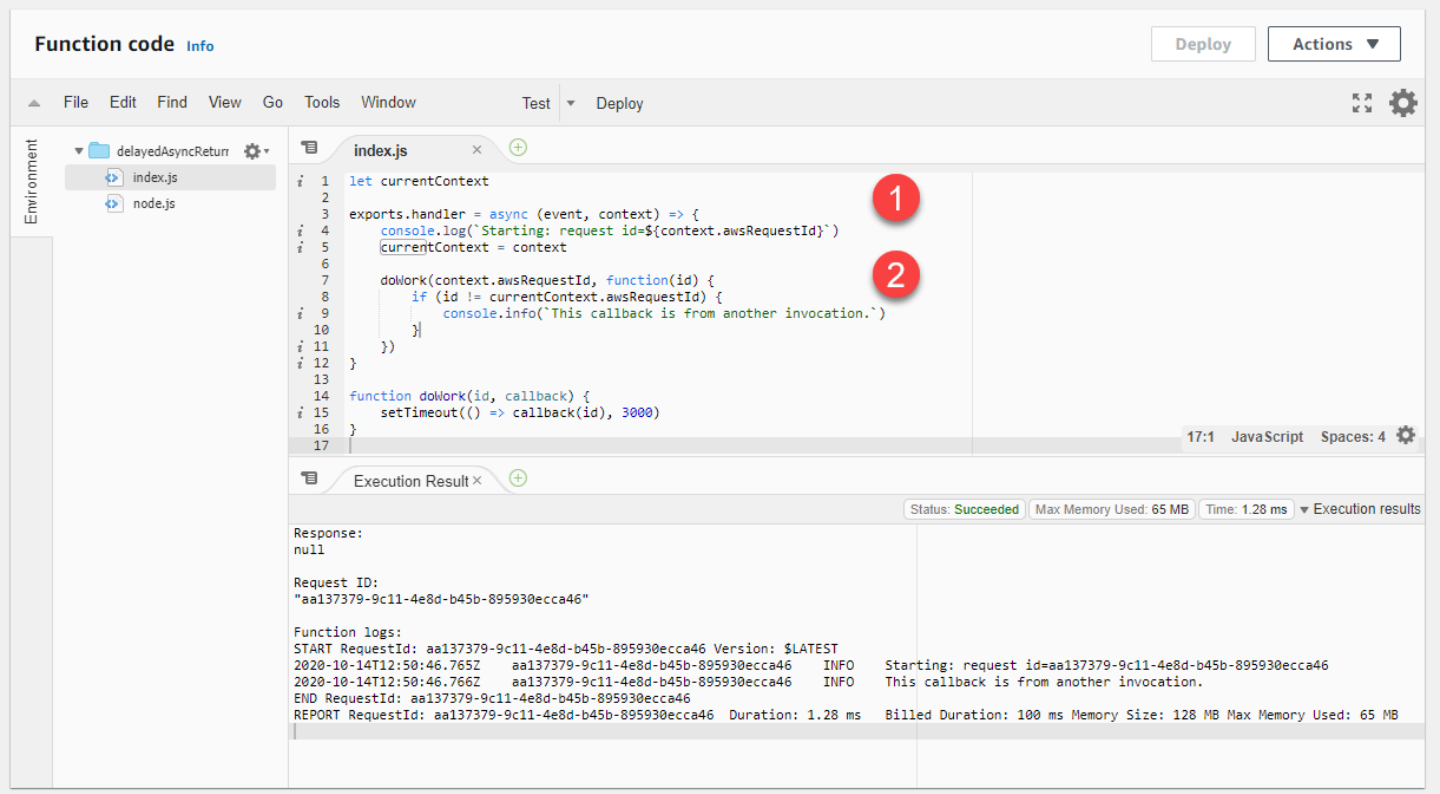


1. Le code appelle la fonction asynchrone `doWork` à l'aide du mot clé await dans le gestionnaire.
2. La fonction `doWork` met un certain temps à s'exécuter avant de résoudre la promesse.
3. La fonction expire, car `doWork` prend plus de temps que le délai imparti et le résultat du rappel n'est pas renvoyé lors d'une invocation ultérieure.

En général, assurez-vous que les processus d'arrière-plan ou les rappels dans le code se terminent avant que l'exécution du code ne prenne fin. Si cela n'est pas possible dans votre cas d'utilisation, vous pouvez utiliser un identifiant pour vous assurer que le rappel appartient à l'invocation en cours. Pour ce faire, vous pouvez utiliser l'identifiant awsRequestId fourni par l'objet de contexte. En transmettant cette valeur au rappel asynchrone, vous pouvez comparer la valeur transmise à la valeur actuelle pour détecter si le rappel provient d'une autre invocation :

```
let currentContext

exports.handler = async (event, context) => {
    console.log(`Starting: request id=$\{context.awsRequestId}`)
    currentContext = context

    doWork(context.awsRequestId, function(id) {
        if (id != currentContext.awsRequestId) {
```

```
            console.info(`This callback is from another invocation.`)
        }
    })

}

function doWork(id, callback) {
    setTimeout(() => callback(id), 3000)

}
```



1. Le gestionnaire de fonctions Lambda utilise le paramètre context, qui donne accès à un ID de requête d'invocation unique.
2. L'`awsRequestId` est transmis à la fonction doWork. Dans le rappel, l'ID est comparé à la valeur `awsRequestId` de l'invocation en cours. Si ces valeurs sont différentes, le code peut agir en conséquence.

Résoudre les problèmes de déploiement dans Lambda

Lorsque vous mettez à jour votre fonction, Lambda déploie la modification en lançant de nouvelles instances de cette fonction avec le code ou les paramètres mis à jour. Les erreurs de déploiement empêchent l'utilisation de la nouvelle version et peuvent résulter de problèmes liés au package de déploiement, au code, à vos autorisations ou à vos outils.

Lorsque vous déployez des mises à jour de votre fonction directement avec l'API Lambda ou avec un client tel que le AWS CLI, vous pouvez voir les erreurs provenant de Lambda directement dans la sortie. Si vous utilisez des services tels que AWS CloudFormation AWS CodeDeploy, ou AWS CodePipeline, recherchez la réponse de Lambda dans les journaux ou le flux d'événements de ce service.

Les rubriques suivantes fournissent des conseils de dépannage pour les erreurs et les problèmes que vous pouvez rencontrer lors de l'utilisation de l'API, de la console ou des outils Lambda. Si vous rencontrez un problème qui n'est pas répertorié ici, vous pouvez utiliser le bouton Commentaire sur cette page pour le signaler.

Pour plus de conseils de dépannage et de réponses aux questions courantes de support, visitez le Centre de connaissances AWS.

Pour plus d'informations sur le débogage et le dépannage des applications Lambda, consultez Débogage dans Serverless Land.

Rubriques

- Général : autorisation refusée/impossible de charger ce fichier
- Général : Une erreur se produit lors de l'appel du UpdateFunctionCode
- Amazon S3 : code d'erreur PermanentRedirect.
- Général : impossible de trouver, impossible de charger, impossible d'importer, classe introuvable, aucun fichier ou répertoire de ce type
- Général : gestionnaire de méthode non défini
- Général : limite de stockage du code Lambda dépassée
- Lambda : échec de la conversion de couche
- Lambda : ou InvalidParameterValueException RequestEntityTooLargeException
- Lambda : InvalidParameterValueException
- Lambda : quotas de simultanéité et de mémoire

- Lambda : configuration d'alias non valide pour la simultanéité provisionnée.

Général : autorisation refusée/impossible de charger ce fichier

Erreur : EACCES : autorisation refusée, ouvrez « var/task/index/.js »

Erreur : impossible de charger ce fichier - cette fonction

Erreur : [Erron 13] Autorisation refusée : var/task/function '/ .py'

Le runtime Lambda a besoin d'une autorisation pour lire les fichiers de votre package de déploiement. Dans la notation octale des autorisations Linux, Lambda a besoin de 644 autorisations pour les fichiers non exécutables (rw-r--r--) et de 755 autorisations () pour les répertoires et les fichiers exécutables. rwxr-xr-x

Sous Linux et macOS, utilisez la commande chmod pour modifier les autorisations de fichiers sur les fichiers et les répertoires de votre package de déploiement. Par exemple, pour octroyer à un fichier non exécutable les autorisations correctes, exécutez la commande suivante.

```
chmod 644 <filepath>
```

Pour modifier les autorisations relatives aux fichiers dans Windows, voir Définir, afficher, modifier ou supprimer des autorisations sur un objet dans la documentation Microsoft Windows.

> **Note**
>
> Si vous n'accordez pas à Lambda les autorisations nécessaires pour accéder aux répertoires de votre package de déploiement, Lambda définit les autorisations pour ces répertoires sur 755 (). rwxr-xr-x

Général : Une erreur se produit lors de l'appel du UpdateFunctionCode

Erreur : une erreur s'est produite (RequestEntityTooLargeException) lors de l'appel de l' UpdateFunctionCode opération

Lorsque vous chargez un package de déploiement ou une archive de couche directement dans Lambda, la taille du fichier ZIP est limitée à 50 Mo. Pour charger un fichier plus volumineux, stockez-le dans Amazon S3, puis utilisez les paramètres S3Bucket et S3Key.

> **Note**
>
> Lorsque vous téléchargez un fichier directement avec le AWS CLI AWS SDK ou autre, le fichier ZIP binaire est converti en base64, ce qui augmente sa taille d'environ 30 %. Pour permettre cela et aussi prendre en compte la taille d'autres paramètres dans la demande, la limite de taille réelle de demande que Lambda applique est supérieure. Pour cette raison, la limite de 50 Mo est approximative.

Amazon S3 : code d'erreur PermanentRedirect.

Erreur : une erreur s'est produite pendant GetObject. Code d'erreur S3 : PermanentRedirect. Message d'erreur S3 : Le compartiment se trouve dans cette région : us-east-2. Veuillez utiliser cette région pour renouveler la demande

Lorsque vous chargez le package de déploiement d'une fonction à partir d'un compartiment Amazon S3, le compartiment doit se trouver dans la même région que la fonction. Ce problème peut se produire lorsque vous spécifiez un objet Amazon S3 dans un appel à UpdateFunctionCode, ou lorsque vous utilisez les commandes de package et de déploiement dans la AWS SAM CLI AWS CLI ou. Créez un compartiment d'artefact de déploiement pour chaque région dans laquelle vous développez des applications.

Général : impossible de trouver, impossible de charger, impossible d'importer, classe introuvable, aucun fichier ou répertoire de ce type

Erreur : module 'function' introuvable

Erreur : impossible de charger ce fichier - cette fonction

Erreur : impossible d'importer le module 'function'

Erreur : Classe introuvable : Function.handler

Erreur fork/exec /var/task/function : aucun fichier ou répertoire de ce type

Erreur : impossible de charger le type 'Function.Handler' à partir de l'assemblage 'Function'.

Le nom du fichier ou de la classe dans la configuration du gestionnaire de votre fonction ne correspond pas à votre code. Consultez la section suivante pour plus d'informations.

Général : gestionnaire de méthode non défini

Erreur : index.handler n'est pas défini ou n'est pas exporté

Erreur : gestionnaire 'handler' manquant dans le module 'function'

Erreur : « gestionnaire » de méthode non défini pour #<:0x000055b76ccebf98> LambdaHandler

Erreur : aucune méthode publique nommée handleRequest avec la signature de méthode appropriée trouvée dans la classe function.Handler

Erreur : méthode 'handleRequest' introuvable dans le type 'Function.Handler' de l'assemblage 'Function'

Le nom de la méthode du gestionnaire dans la configuration du gestionnaire de votre fonction ne correspond pas à votre code. Chaque environnement d'exécution définit une convention de dénomination pour les gestionnaires, telle que*`filename`*.*`methodname`*. Le gestionnaire correspond à la méthode dans le code de votre fonction que l'environnement d'exécution exécute lorsque la fonction est invoquée.

Pour certains langages, Lambda fournit une bibliothèque avec une interface pour laquelle une méthode de gestionnaire doit avoir un nom spécifique. Pour plus d'informations sur l'attribution de noms de gestionnaire pour chaque langue, consultez les rubriques suivantes.

- Création de fonctions Lambda avec Node.js
- Création de fonctions Lambda avec Python
- Création de fonctions Lambda avec Ruby
- Création de fonctions Lambda avec Java
- Création de fonctions Lambda avec Go
- Création de fonctions Lambda avec C#
- Création de fonctions Lambda avec PowerShell

Général : limite de stockage du code Lambda dépassée

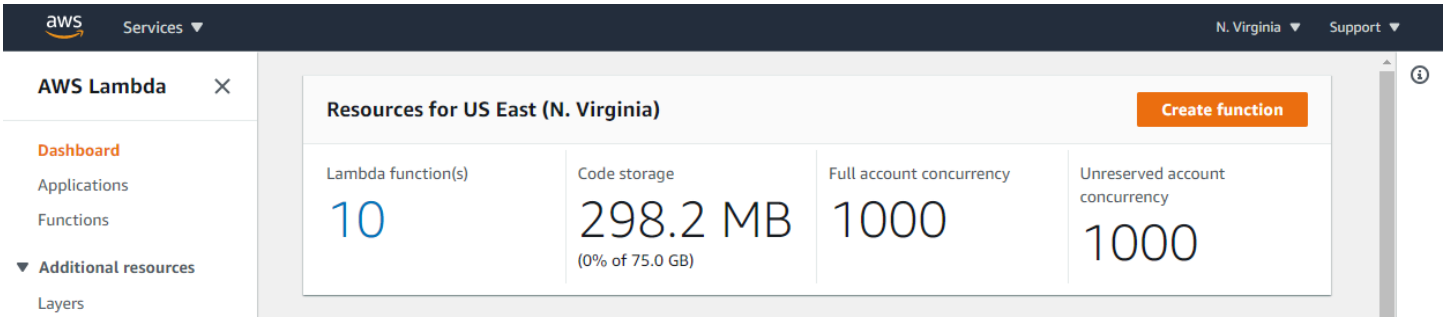
Erreur : limite de stockage de code dépassée.

Lambda stocke le code de votre fonction dans un compartiment S3 interne réservé à votre compte. Chaque AWS compte dispose de 75 Go de stockage dans chaque région. Le stockage de code inclut

le stockage total utilisé à la fois par les fonctions et les couches Lambda. Si vous atteignez le quota, vous recevez un CodeStorageExceededExceptionlorsque vous tentez de déployer de nouvelles fonctions.

Gérez l'espace de stockage disponible en nettoyant les anciennes versions des fonctions, en supprimant le code inutilisé ou en utilisant des couches Lambda. En outre, il est recommandé d'utiliser des AWS comptes distincts pour des charges de travail distinctes afin de gérer les quotas de stockage.

Vous pouvez consulter votre utilisation totale du stockage dans la console Lambda, dans le sous-menu Tableau de bord :



Lambda : échec de la conversion de couche

Erreur : la conversion de la couche Lambda a échoué. Pour obtenir des conseils sur la résolution de ce problème, consultez la page Résolution des problèmes de déploiement dans Lambda du Guide de l'utilisateur Lambda.

Lorsque vous configurez une fonction Lambda avec une couche, Lambda fusionne la couche avec le code de votre fonction. Si ce processus échoue, Lambda renvoie cette erreur. Si vous rencontrez cette erreur, réalisez les actions suivantes :

- Supprimer tous les fichiers inutiles de votre couche
- Supprimer tous les liens symboliques de votre couche
- Renommer tous les fichiers qui ont le même nom qu'un répertoire dans l'une des couches de votre fonction

Lambda : ou InvalidParameterValueException RequestEntityTooLargeException

Erreur InvalidParameterValueException : Lambda n'a pas pu configurer vos variables d'environnement car les variables d'environnement que vous avez fournies ont dépassé la limite de 4 Ko. Chaîne mesurée : {"A1" : « u SFe Y5 cyPiPn 7atNx5bSM...

Erreur RequestEntityTooLargeException : la demande doit être inférieure à 5120 octets pour l'opération UpdateFunctionConfiguration

La taille maximale de l’objet variable stocké dans la configuration de la fonction ne doit pas dépasser 4096 octets. Cela inclut les noms de clés, les valeurs, les guillemets, les virgules et les crochets. La taille totale du corps de requête HTTP est également limitée.

```
{
    "FunctionName": "my-function",
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
    "Runtime": "nodejs24.x",
    "Role": "arn:aws:iam::123456789012:role/lambda-role",
    "Environment": {
        "Variables": {
            "BUCKET": "amzn-s3-demo-bucket",
            "KEY": "file.txt"
        }
    },
    ...
}
```

Dans cet exemple, l’objet comporte 39 caractères et utilise 39 octets lorsqu’il est stocké (sans espace) comme chaîne `{"BUCKET":"amzn-s3-demo-bucket","KEY":"file.txt"}`. Les caractères ASCII standard dans les valeurs des variables d’environnement utilisent un octet chacun. Les caractères ASCII et Unicode étendus peuvent utiliser entre 2 et 4 octets par caractère.

Lambda : InvalidParameterValueException

Erreur InvalidParameterValueException : Lambda n'a pas pu configurer vos variables d'environnement car les variables d'environnement que vous avez fournies contiennent des clés réservées dont la modification n'est actuellement pas prise en charge.

Lambda réserve certaines clés de variables d’environnement pour une utilisation interne. Par exemple, `AWS_REGION` est utilisé par l’environnement d’exécution pour déterminer la région actuelle

et ne peut pas être remplacé. D'autres variables, comme PATH, sont utilisées par l'environnement d'exécution, mais peuvent être étendues dans la configuration de votre fonction. Pour obtenir une liste complète, veuillez consulter Variables d'environnement d'exécution définies.

Lambda : quotas de simultanéité et de mémoire

Erreur : la fonction spécifiée ConcurrentExecutions réduit le compte en UnreservedConcurrentExecution dessous de sa valeur minimale

Erreur : la valeur MemorySize « » n'a pas satisfait la contrainte : le membre doit avoir une valeur inférieure ou égale à 3008

Ces erreurs se produisent lorsque vous dépassez les quotas de simultanéité ou de mémoire pour votre compte. AWS Les nouveaux comptes ont réduit la simultanéité et les quotas de mémoire. Pour résoudre les erreurs liées à la simultanéité, vous pouvez demander une augmentation des quotas. Vous ne pouvez pas demander d'augmentation des quotas de mémoire.

- Simultanéité : vous pouvez recevoir un message d'erreur si vous essayez de créer une fonction à l'aide de la simultanéité réservée ou provisionnée, ou si votre demande de simultanéité par fonction (PutFunctionConcurrency) excède le quota de simultanéité de votre compte.
- Mémoire : des erreurs surviennent lorsque la capacité de mémoire allouée à la fonction dépasse le quota de mémoire de votre compte.

Lambda : configuration d'alias non valide pour la simultanéité provisionnée.

Erreur : configuration d'alias non valide pour la simultanéité provisionnée

Cette erreur se produit lorsque vous essayez de mettre à jour le code ou la configuration d'une fonction Lambda alors qu'un alias doté d'une simultanéité provisionnée pointe vers une version présentant des problèmes. Lambda pré-initialise les environnements d'exécution pour une simultanéité provisionnée, et si ces environnements ne peuvent pas être correctement initialisés en raison d'erreurs de code, de contraintes de ressources ou de l'utilisation d'une pile ou d'un alias affecté, le déploiement échoue. Si vous rencontrez ce problème, réalisez les actions suivantes :

1. Restaurer l'alias : mettez temporairement à jour l'alias pour qu'il pointe vers la version qui fonctionnait auparavant.

```
aws lambda update-alias \
 --function-name <function-name> \
```

```
  --name <alias-name> \
  --function-version <known-good-version>
```

2. Corriger le code d'initialisation Lambda : assurez-vous que le code d'initialisation exécuté en dehors du gestionnaire ne contient aucune exception non détectée et initialisez les clients et les connexions.
3. Redéployer de façon sûre : déployez le code corrigé et publiez une nouvelle version. Mettez ensuite à jour l'alias pour qu'il pointe vers la version corrigée. Réactivez éventuellement la simultanéité provisionnée, si nécessaire.

Si vous l'utilisez AWS CloudFormation, mettez à jour la définition de la pile `FunctionVersion:!GetAtt version.Version` afin que l'alias pointe vers la version de travail :

```
alias:
 Type: AWS::Lambda::Alias
 Properties:
 FunctionName: !Ref function
FunctionVersion: !GetAtt version.Version
 Name: BLUE
 ProvisionedConcurrencyConfig:
 ProvisionedConcurrentExecutions: 1
```

Résoudre les problèmes d'invocation dans Lambda

Lorsque vous invoquez une fonction Lambda, Lambda valide la demande et vérifie la capacité de dimensionnement avant d'envoyer l'événement à votre fonction ou, pour une invocation asynchrone, à la file d'attente d'événements. Les erreurs d'invocation peuvent être causées par des problèmes de paramètres de demande, de structure d'événement, de paramètres de fonction, d'autorisations utilisateur, d'autorisations de ressources ou de limites.

Si vous invoquez la fonction directement, les erreurs d'invocation sont visibles dans la réponse de Lambda. Si vous invoquez la fonction de manière asynchrone avec un mappage de source d'événement ou via un autre service, vous pouvez trouver des erreurs dans les journaux, une file d'attente de lettres mortes ou une destination en cas d'échec. Les options de gestion des erreurs et le comportement des nouvelles tentatives varient en fonction de la façon dont vous invoquez la fonction et du type d'erreur.

Pour obtenir la liste des types d'erreur que l'opération `Invoke` peut renvoyer, consultez Invoquer.

Rubriques

- Lambda : expiration de la fonction pendant la phase d'initialisation (Sandbox.Timedout)
- IAM : lambda : non autorisé InvokeFunction
- Lambda : Impossible de trouver un bootstrap valide (Runtime). InvalidEntrypoint)
- Lambda : L'opération ne peut pas être effectuée ResourceConflictException
- Lambda : La fonction est bloquée à l'état Pending (En attente)
- Lambda : Une fonction utilise toute la simultanéité
- Général : Impossible d'invoquer la fonction avec d'autres comptes ou services
- Général : L'invocation de la fonction boucle
- Lambda : Routage d'alias avec une simultanéité approvisionnée
- Lambda : Démarrages à froid avec la simultanéité allouée
- Lambda : Démarrages à froid avec de nouvelles versions
- Lambda : sortie inattendue de Node.js pendant l'exécution (Runtime). NodejsExit)
- EFS : La fonction n'a pas pu monter le système de fichiers EFS
- EFS : La fonction n'a pas pu se connecter au système de fichiers EFS
- EFS : La fonction n'a pas pu monter le système de fichiers EFS en raison d'une expiration de délai
- Lambda : Lambda a détecté un processus d'E/S qui prenait trop de temps
- Conteneur : CodeArtifactUserException erreurs
- Conteneur : InvalidEntrypoint erreurs

Lambda : expiration de la fonction pendant la phase d'initialisation (Sandbox.Timedout)

Erreur : Task timed out after 3.00 seconds

Lorsque la phase d'initialisation expire, Lambda initialise à nouveau l'environnement d'exécution en relançant la phase `Init` lorsque la prochaine demande d'invocation arrive. Cela s'appelle une init supprimée. Toutefois, si votre fonction est configurée avec un délai d'expiration court (généralement environ 3 secondes), l'initialisation supprimée risque de ne pas se terminer pendant le délai imparti, ce qui provoquera une nouvelle expiration de la phase `Init`. Sinon, l'initialisation supprimée se termine mais ne laisse pas suffisamment de temps à la phase d'invocation pour se terminer, ce qui entraîne l'expiration de la phase `Invoke`.

Pour réduire les erreurs d'expiration, utilisez l'une des stratégies suivantes :

- Augmentez la durée du délai d'expiration de la fonction : prolongez le délai d'expiration pour donner aux phases `Invoke` et `Init` et le temps de se terminer.
- Augmentez l'allocation de mémoire de la fonction : plus de mémoire signifie également une allocation plus proportionnelle du processeur, ce qui peut accélérer à la fois les phases `Invoke` et `Init`.
- Optimisez le code d'initialisation de la fonction : réduisez le temps nécessaire à l'initialisation afin de garantir que les phases `Init` et `Invoke` puissent se terminer dans le délai configuré.

IAM : lambda : non autorisé InvokeFunction

Erreur : L'utilisateur : arn:aws:iam : :123456789012 : l'utilisateur/développeur n'est pas autorisé à exécuter : lambda : on resource : my-function InvokeFunction

Votre utilisateur, ou le rôle que vous assumez, doit avoir une autorisation pour invoquer une fonction. Cette exigence s'applique également aux fonctions Lambda et à d'autres ressources de calcul qui invoquent des fonctions. Ajoutez le AWSLambdarôle de stratégie AWS géré à votre utilisateur ou ajoutez une politique personnalisée qui autorise l'`lambda:InvokeFunction`action sur la fonction cible.

> **Note**
>
> Le nom de l'action IAM (`lambda:InvokeFunction`) fait référence à l'opération de l'API Lambda `Invoke`.

Pour plus d'informations, consultez Gestion des autorisations dans AWS Lambda.

Lambda : Impossible de trouver un bootstrap valide (Runtime). InvalidEntrypoint)

Erreur : Impossible de trouver un ou plusieurs bootstrap valides : [/var/task/bootstrap /opt/bootstrap]

Cette erreur se produit généralement lorsque la racine de votre package de déploiement ne contient pas de fichier exécutable nommé `bootstrap`. Par exemple, si vous déployez une fonction `provided.al2023` avec un fichier .zip, le fichier `bootstrap` doit se trouver à la racine du fichier .zip, et non dans un répertoire.

Lambda : L'opération ne peut pas être effectuée ResourceConflictException

Erreur ResourceConflictException : Impossible d'effectuer l'opération pour le moment. La fonction est actuellement dans l'état suivant : En attente

Lorsque vous connectez une fonction à un cloud privé virtuel (Virtual Private Cloud, VPC) au moment de la création, la fonction passe à l'état `Pending` pendant que Lambda crée des interfaces réseau Elastic. Pendant ce temps, vous ne pouvez pas invoquer ni modifier votre fonction. Si vous connectez votre fonction à un VPC après la création, vous pouvez l'invoquer pendant que la mise à jour est en attente, mais vous ne pouvez pas modifier son code ni sa configuration.

Pour plus d'informations, consultez États de la fonction Lambda.

Lambda : La fonction est bloquée à l'état Pending (En attente)

Erreur : Une fonction est bloquée à l'état *`Pending`* depuis plusieurs minutes.

Si une fonction est bloquée à l'état `Pending` depuis plus de six minutes, appelez l'une des opérations d'API suivantes pour la débloquer :

- UpdateFunctionCode
- UpdateFunctionConfiguration
- PublishVersion

Lambda annule l'opération en attente et met la fonction à l'état `Failed`. Vous pouvez ensuite essayer une autre mise à jour.

Lambda : Une fonction utilise toute la simultanéité

Problème : une fonction utilise toute la simultanéité disponible, limitant ainsi d'autres fonctions.

Pour diviser la simultanéité disponible de votre AWS compte dans une AWS région en groupes, utilisez la simultanéité réservée. La simultanéité réservée garantit qu'une fonction s'adapte toujours à la simultanéité qui lui est assignée et qu'elle n'utilise pas plus de simultanéité que ce qui lui est attribué.

Général : Impossible d'invoquer la fonction avec d'autres comptes ou services

Problème : vous pouvez invoquer la fonction directement, mais elle ne s'exécute pas lorsqu'elle est invoquée par un autre service ou compte.

Vous accordez à d'autres services et comptes l'autorisation d'invoquer une fonction dans la stratégie basée sur les ressourcesde la fonction. Si l'appelant se trouve dans un autre compte, cet utilisateur doit également avoir l'autorisation d'invoquer des fonctions.

Général : L'invocation de la fonction boucle

Problème : La fonction est invoquée en continu dans une boucle.

Cela se produit généralement lorsque votre fonction gère les ressources du même AWS service qui la déclenche. Par exemple, il est possible de créer une fonction qui stocke un objet dans un compartiment Amazon Simple Storage Service (Amazon S3) configuré avec une notification qui invoque à nouveau la fonction. Pour arrêter l'exécution de la fonction, réduisez la simultanéité disponible de votre fonction à zéro, ce qui limite toutes les invocations futures. Identifiez ensuite le chemin de code ou l'erreur de configuration qui a provoqué l'invocation récursive. Lambda détecte et arrête automatiquement les boucles récursives pour certains AWS services et. SDKs Pour de plus amples informations, veuillez consulter the section called "Détection de boucle récursive".

Lambda : Routage d'alias avec une simultanéité approvisionnée

Problème: invocations de débordement de simultanéité approvisionnée pendant le routage d'alias.

Lambda utilise un modèle probabiliste simple pour distribuer le trafic entre les deux versions de la fonction. Quand le niveau de trafic est faible, il se peut que vous observiez une variance élevée entre les pourcentages de trafic configuré et réel sur chaque version. Si votre fonction utilise une simultanéité approvisionnée, vous pouvez éviter des invocations de débordement en configurant un plus grand nombre d'instances de simultanéité approvisionnées pendant que le routage d'alias est actif.

Lambda : Démarrages à froid avec la simultanéité allouée

Problème : Vous remarquez des démarrages à froid après avoir activé la simultanéité allouée.

Lorsque le nombre d'exécutions simultanées sur une fonction est inférieur ou égal au niveau de simultanéité configuré, il ne devrait pas y avoir de démarrage à froid. Pour vous aider à vérifier le bon fonctionnement de la simultanéité, procédez comme suit :

- Vérifiez que la simultanéité allouée est activée sur la version de fonction ou l'alias.

  > **Note**
  >
  > La simultanéité allouée n'est pas configurable sur la version non publiée de la fonction ($LATEST).

- Assurez-vous que vos déclencheurs invoquent la version ou l'alias de fonction correct. Par exemple, si vous utilisez Amazon API Gateway, vérifiez qu'API Gateway invoque la version de fonction ou l'alias avec la simultanéité approvisionnée, et non $LATEST. Pour confirmer que la simultanéité provisionnée est utilisée, vous pouvez consulter la métrique ProvisionedConcurrencyInvocations Amazon CloudWatch . Une valeur non nulle indique que la fonction traite des invocations sur les environnements d'exécution initialisés.
- Déterminez si la simultanéité de vos fonctions dépasse le niveau configuré de simultanéité provisionnée en vérifiant la métrique. ProvisionedConcurrencySpilloverInvocations CloudWatch Une valeur différente de zéro indique que toute la simultanéité allouée est en cours d'utilisation et qu'une invocation s'est produite avec un démarrage à froid.
- Vérifiez la fréquence des invocations (demandes par seconde). Les fonctions avec simultanéité allouée ont un taux maximal de 10 demandes par seconde par simultanéité allouée. Par exemple, une fonction configurée avec une simultané allouée de 100 peut traiter 1 000 demandes par seconde. Si le taux d'invocations dépasse 1 000 requêtes par seconde, certains démarrages à froid peuvent se produire.

Lambda : Démarrages à froid avec de nouvelles versions

Problème : Vous notez des démarrages à froid lors du déploiement de nouvelles versions de votre fonction.

Lorsque vous mettez à jour un alias de fonction, Lambda déplace automatiquement la simultanéité approvisionnée vers la nouvelle version en fonction des pondérations configurées sur l'alias.

Erreur : KMSDisabled exception : Lambda n'a pas pu déchiffrer les variables d'environnement car la clé KMS utilisée est désactivée. Veuillez vérifier les paramètres de clé KMS de la fonction.

Cette erreur peut se produire si votre clé AWS Key Management Service (AWS KMS) est désactivée ou si l'autorisation permettant à Lambda d'utiliser la clé est révoquée. Si l'octroi est manquant, configurez la fonction pour utiliser une autre clé. Ensuite, réaffectez la clé personnalisée pour recréer l'octroi.

Lambda : sortie inattendue de Node.js pendant l'exécution (Runtime). NodejsExit)

Problème : le client d'exécution Lambda a détecté un code de sortie Node.js inattendu.

Cette erreur se produit lorsque votre fonction se ferme avant que toutes les promesses ne soient réglées, par exemple en raison d'un bogue de code. Cela peut également se produire lorsque Node.js détecte un blocage empêchant le règlement des promesses. Cette erreur affecte uniquement les gestionnaires de style asynchrone, pas les gestionnaires de style rappel.

Runtimes concernés : Node.js 18 et versions ultérieures.

Pour résoudre ce problème, procédez comme suit :

1. Vérifiez que le code de votre fonction ne contient pas de promesses non réglées dans les gestionnaires asynchrones.
2. Assurez-vous que toutes les promesses sont correctement réglées (résolues ou rejetées) avant la fin de la fonction.
3. Vérifiez votre code pour détecter les conditions de course potentielles lors d'opérations asynchrones.

Pour plus d'informations sur les codes de sortie de Node.js et l'arrêt des processus, consultez la documentation Node.js.

EFS : La fonction n'a pas pu monter le système de fichiers EFS

Erreur EFSMount FailureException : La fonction n'a pas pu monter le système de fichiers EFS avec le point d'accès arn:aws:elasticfilesystem:us-east- 2:123456789012:access-point/fsap-015cxmplb72b405fd.

La demande de montage sur le système de fichiers de la fonction a été rejetée. Vérifiez les autorisations de la fonction et confirmez que son système de fichiers et son point d'accès existent et sont prêts à l'emploi.

EFS : La fonction n’a pas pu se connecter au système de fichiers EFS

Erreur EFSMount ConnectivityException : La fonction n'a pas pu se connecter au système de fichiers Amazon EFS avec le point d'accès arn:aws:elasticfilesystem:us-east- 2:123456789012:access-point/fsap-015cxmplb72b405fd. Vérifiez la configuration de votre réseau et réessayez.

La fonction n’a pas pu établir de connexion au système de fichiers de la fonction avec le protocole NFS (port TCP 2049). Vérifiez le groupe de sécurité et la configuration de routage pour les sous-réseaux du VPC.

Si vous rencontrez ces erreurs après avoir mis à jour les paramètres de configuration VPC de votre fonction, essayez de démonter puis de remonter le système de fichiers.

EFS : La fonction n’a pas pu monter le système de fichiers EFS en raison d’une expiration de délai

Erreur EFSMount TimeoutException : La fonction n'a pas pu monter le système de fichiers EFS avec le point d'accès {arn:aws:elasticfilesystem:us-east- 2:123456789012:access-point/fsap-015cxmplb72b405fd} en raison d'un délai de montage dépassé.

La fonction a pu se connecter au système de fichiers de la fonction, mais l’opération de montage a expiré. Réessayez après un court laps de temps et envisagez de limiter la concurrence de la fonction pour réduire la charge sur le système de fichiers.

Lambda : Lambda a détecté un processus d’E/S qui prenait trop de temps

EFSIOException: Cette instance de fonction a été arrêtée car Lambda a détecté un processus d'E/S trop long.

Une invocation précédente a expiré et Lambda n’a pas pu mettre fin à l’exécution du gestionnaire de fonction. Ce problème peut se produire lorsqu’un système de fichiers joint manque de crédits de rafale et que le débit de base est insuffisant. Pour augmenter le débit, vous pouvez augmenter la taille du système de fichiers ou utiliser le débit provisionné.

Conteneur : CodeArtifactUserException erreurs

Erreur : message CodeArtifactUserPendingException d'erreur

L'optimisation CodeArtifact est en attente. La fonction passera à l’état Actif lorsque Lambda aura terminé l’optimisation. Code de réponse HTTP : 409.

Erreur : message CodeArtifactUserDeletedException d'erreur

CodeArtifact Il est prévu de le supprimer. Code de réponse HTTP : 409.

Erreur : message CodeArtifactUserFailedException d'erreur

Lambda n'a pas réussi à optimiser le code. Vous devez corriger le code et le charger à nouveau. Code de réponse HTTP : 409.

Conteneur : InvalidEntrypoint erreurs

Erreur : Runtime. ExitError ou « ErrorType » : « Runtime. InvalidEntrypoint«

Vérifiez que l'ENTRYPOINT de votre image de conteneur inclut le chemin absolu comme emplacement. Vérifiez également que l'image ne contient pas de lien symbolique en tant qu'ENTRYPOINT.

Erreur : vous utilisez un CloudFormation modèle et votre conteneur ENTRYPOINT est remplacé par une valeur nulle ou vide.

Passez en revue la ImageConfigressource dans le CloudFormation modèle. Si vous déclarez une ressource `ImageConfig` dans votre modèle, vous devez fournir des valeurs non vides pour les trois propriétés.

Résoudre les problèmes d’exécution dans Lambda

Lorsque le runtime Lambda exécute le code de fonction, l’événement peut être traité sur une instance de la fonction qui traite déjà les événements depuis un certain temps, ou nécessiter l’initialisation d’une nouvelle instance. Des erreurs peuvent se produire lors de l’initialisation de la fonction, lorsque le code de gestionnaire traite l’événement ou lorsque la fonction renvoie (ou ne parvient pas à renvoyer) une réponse.

Les erreurs d’exécution de fonction peuvent être causées par des problèmes de code, de configuration de fonction, de ressources en aval ou d’autorisations. Si vous appelez la fonction directement, les erreurs de fonction sont visibles dans la réponse de Lambda. Si vous appelez la fonction de manière asynchrone, avec un mappage de source d’événement ou via un autre service, vous pouvez trouver les erreurs dans les journaux, une file d’attente de lettres mortes ou une destination réservées aux échecs. Les options de gestion des erreurs et le comportement des nouvelles tentatives varient en fonction de la façon dont vous appelez la fonction et du type d’erreur.

Lorsque le code de fonction ou le runtime Lambda renvoient une erreur, le code d'état indiqué dans la réponse de Lambda est 200 OK. La présence d'une erreur dans la réponse est indiquée par un en-tête nommé `X-Amz-Function-Error`. Les codes d'état des séries 400 et 500 sont réservés aux erreurs d'invocation.

Rubriques

- Lambda : débogage à distance avec Visual Studio Code
- Lambda : l'exécution prend trop de temps
- Lambda : données utiles d'événement imprévu
- Lambda : données utiles plus volumineuses qu'attendu
- Lambda : erreurs de codage et de décodage JSON
- Lambda : les journaux ou les traces n'apparaissent pas
- Lambda : certains journaux de ma fonction n'apparaissent pas
- Lambda : la fonction renvoie avant la fin de l'exécution
- Lambda : exécution d'une version ou d'un alias de fonction imprévu
- Lambda : détection de boucles infinies
- Général : indisponibilité du service en aval
- AWS SDK : versions et mises à jour
- Python : les bibliothèques se chargent de manière incorrecte
- Java : votre fonction met plus de temps à traiter les événements après la mise à jour de Java 11 vers Java 17
- Kafka : Problèmes de gestion des erreurs et de configuration des nouvelles tentatives

Lambda : débogage à distance avec Visual Studio Code

Problème : difficulté à résoudre les problèmes liés au comportement complexe des fonctions Lambda dans l'environnement réel AWS

Lambda fournit une fonctionnalité de débogage à distance via AWS Toolkit for Visual Studio Code. Pour instructions générales et de configuration, consultez Déboguer à distance des fonctions Lambda avec Visual Studio Code.

Pour obtenir des instructions détaillées sur le dépannage, les cas d'utilisation avancés et la disponibilité régionale, consultez la section Débogage à distance des fonctions Lambda dans AWS Toolkit for Visual Studio Code le guide de l'utilisateur.

Lambda : l'exécution prend trop de temps

Problème : l'exécution de la fonction prend trop de temps.

Si l'exécution de votre code est beaucoup plus longue dans Lambda que sur votre ordinateur local, cela peut être dû à une limite au niveau de la mémoire ou de la puissance de traitement disponible pour la fonction. Configurez la fonction avec de la mémoire supplémentaire pour augmenter à la fois la mémoire et la capacité d'UC.

Lambda : données utiles d'événement imprévu

Problème : erreurs de fonctionnement liées à du JSON mal formé ou à une validation inadéquate des données.

Toutes les fonctions Lambda reçoivent des données utiles d'événement dans le premier paramètre du gestionnaire. Les données utiles de l'événement sont une structure JSON qui peut contenir des tableaux et des éléments imbriqués.

Un JSON mal formé peut se produire lorsqu'il est fourni par des services en amont qui n'utilisent pas de processus robuste pour vérifier les structures JSON. Cela se produit lorsque les services concatènent des chaînes de texte ou intègrent des entrées utilisateur qui n'ont pas été nettoyées. Le JSON est également fréquemment sérialisé pour le transfert d'un service à l'autre. Analysez toujours les structures JSON en tant que producteur et consommateur de JSON pour vous assurer que la structure est valide.

De même, le fait de ne pas vérifier les plages de valeurs dans les données utiles de l'événement peut entraîner des erreurs. L'exemple suivant montre une fonction qui calcule une retenue d'impôt :

```
exports.handler = async (event) => {
    let pct = event.taxPct
    let salary = event.salary

    // Calculate % of paycheck for taxes
    return (salary * pct)
}
```

Cette fonction utilise un salaire et un taux d'imposition issus des données utiles de l'événement pour effectuer le calcul. Cependant, le code ne vérifie pas si les attributs sont présents. Il ne vérifie pas non plus les types de données ou ne garantit pas les limites, par exemple en s'assurant que le

pourcentage de taxe est compris entre 0 et 1. Par conséquent, les valeurs situées en dehors de ces limites produisent des résultats absurdes. Un type incorrect ou un attribut manquant provoque une erreur d'exécution.

Créez des tests pour vous assurer que votre fonction gère des données utiles plus importantes. La taille maximale des données utiles d'un événement Lambda est de 1 Mo. Selon le contenu, des données utiles plus importantes peuvent entraîner la transmission d'un plus grand nombre d'éléments à la fonction ou l'intégration d'un plus grand nombre de données binaires dans un attribut JSON. Dans les deux cas, cela peut entraîner un traitement supplémentaire pour une fonction Lambda.

Des données utiles plus importantes peuvent également entraîner des délais d'expiration. Par exemple, une fonction Lambda traite un enregistrement toutes les 100 ms et son délai d'expiration est de 3 secondes. Le traitement est réussi pour 0 à 29 éléments des données utiles. Cependant, une fois que les données utiles contiennent plus de 30 éléments, la fonction expire et génère une erreur. Pour éviter cela, assurez-vous que les délais sont définis de manière à gérer le temps de traitement supplémentaire pour le nombre maximal d'éléments attendus.

Lambda : données utiles plus volumineuses qu'attendu

Problème : les fonctions expirent ou provoquent des erreurs en raison de données utiles volumineuses.

Des données utiles plus importantes peuvent entraîner des expirations de délai et des erreurs. Nous vous recommandons de créer des tests pour vous assurer que votre fonction gère les données utiles attendues les plus importantes et vous assurer que le délai d'expiration de la fonction est correctement défini.

Par ailleurs, certaines données utiles d'événements peuvent contenir des pointeurs vers d'autres ressources. Par exemple, une fonction Lambda dotée de 128 Mo de mémoire peut effectuer le traitement d'image sur un fichier JPG stocké sous forme d'objet dans S3. La fonction fonctionne comme prévu avec des fichiers image plus petits. Toutefois, lorsqu'un fichier JPG plus volumineux est fourni en entrée, la fonction Lambda génère une erreur en raison d'un manque de mémoire. Pour éviter cela, les cas de test doivent inclure des exemples tirés des limites supérieures des tailles de données attendues. Le code doit également valider les tailles des données utiles.

Lambda : erreurs de codage et de décodage JSON

Problème : NoSuchKey exception lors de l'analyse des entrées JSON.

Vérifiez que vous traitez correctement les attributs JSON. Par exemple, pour les événements générés par S3, l'attribut `s3.object.key` contient un nom de clé d'objet codé par URL. De nombreuses fonctions traitent cet attribut sous forme de texte pour charger l'objet S3 référencé :

Example

```
const originalText = await s3.getObject({
  Bucket: event.Records[0].s3.bucket.name,
  Key: event.Records[0].s3.object.key
}).promise()
```

Ce code fonctionne avec le nom de clé `james.jpg`, mais génère une erreur `NoSuchKey` lorsque le nom est `james beswick.jpg`. Étant donné que le codage URL convertit les espaces et les autres caractères d'un nom de clé, vous devez vous assurer que les fonctions décodent les clés avant d'utiliser ces données :

Example

```
const originalText = await s3.getObject({
  Bucket: event.Records[0].s3.bucket.name,
  Key: decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, " "))
}).promise()
```

Lambda : les journaux ou les traces n'apparaissent pas

Problème : les journaux n'apparaissent pas dans CloudWatch les journaux.

Problème : les traces n'apparaissent pas dans AWS X-Ray.

Votre fonction doit être autorisée à appeler CloudWatch Logs and X-Ray. Mettez à jour son rôle d'exécution pour lui accorder l'autorisation. Ajoutez les stratégies gérées suivantes pour activer les journaux et le suivi.

- AWSLambdaBasicExecutionRole
- AWSXRayDaemonWriteAccess

Lorsque vous ajoutez des autorisations à votre fonction, modifiez légèrement son code ou sa configuration. Cela force l'arrêt et le remplacement des instances en cours d'exécution de votre fonction, qui ont des informations d'identification obsolètes.

> **Note**
>
> L’affichage des journaux après l’invocation d’une fonction peut prendre de 5 à 10 minutes .

Lambda : certains journaux de ma fonction n’apparaissent pas

Problème : les journaux des fonctions sont absents dans CloudWatch les journaux, même si mes autorisations sont correctes

Si vous atteignez Compte AWS les limites de quota de CloudWatch journalisation, CloudWatch les régulateurs fonctionnent à la journalisation. Dans ce cas, certains journaux générés par vos fonctions peuvent ne pas apparaître dans CloudWatch les journaux.

Si votre fonction produit des journaux à un taux trop élevé pour que Lambda puisse les traiter, cela peut également empêcher les sorties des journaux d'apparaître dans CloudWatch les journaux. Lorsque Lambda ne parvient pas à envoyer des journaux CloudWatch au rythme auquel votre fonction les produit, il supprime les journaux pour empêcher l'exécution de votre fonction de ralentir. Attendez-vous à observer régulièrement la perte de journaux lorsque votre débit de journaux dépasse 2 MB/s pour un seul flux de journaux.

Si votre fonction est configurée pour utiliser des journaux au format JSON, Lambda essaie d'envoyer `logsDropped`un événement CloudWatch à Logs lorsqu'il supprime des journaux. Toutefois, lorsque vous CloudWatch limitez la journalisation de votre fonction, cet événement risque de ne pas atteindre les CloudWatch journaux. Vous ne verrez donc pas toujours d'enregistrement lorsque Lambda supprime les journaux.

Pour vérifier si votre quota de CloudWatch logs Compte AWS est atteint, procédez comme suit :

1. Ouvrez la console Service Quotas.
2. Dans le panneau de navigation, choisissez Services AWS .
3. Dans la liste des AWS services, recherchez Amazon CloudWatch Logs.
4. Dans la liste des quotas de service, choisissez les quotas `CreateLogGroup throttle limit in transactions per second`, `CreateLogStream throttle limit in transactions per second` et `PutLogEvents throttle limit in transactions per second` pour afficher votre utilisation.

Vous pouvez également configurer des CloudWatch alarmes pour vous avertir lorsque l'utilisation de votre compte dépasse la limite que vous avez spécifiée pour ces quotas. Voir Création d'une CloudWatch alarme basée sur un seuil statique pour en savoir plus.

Si les limites de quota par défaut pour CloudWatch les journaux ne sont pas suffisantes pour votre cas d'utilisation, vous pouvez demander une augmentation du quota.

Lambda : la fonction renvoie avant la fin de l'exécution

Problème : (Node.js) la fonction renvoie un objet avant la fin de l'exécution du code

De nombreuses bibliothèques, y compris le AWS SDK, fonctionnent de manière asynchrone. Lorsque vous effectuez un appel réseau ou toute autre opération nécessitant l'attente d'une réponse, les bibliothèques renvoient un objet appelé promesse, qui suit la progression de l'opération en arrière-plan.

Pour attendre que la promesse soit résolue en réponse, utilisez le mot-clé `await`. Celui-ci empêche le code de gestionnaire de s'exécuter jusqu'à ce que la promesse soit résolue en objet contenant la réponse. Si vous n'avez pas besoin d'utiliser les données de la réponse dans votre code, vous pouvez retourner la promesse directement à l'environnement d'exécution.

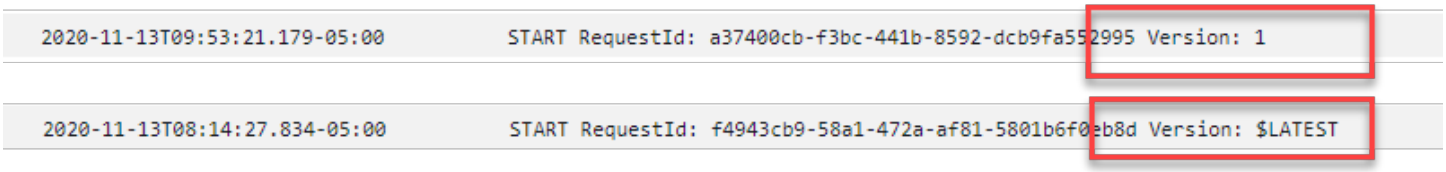
Certaines bibliothèques ne retournent pas de promesses, mais peuvent être enveloppées dans du code qui le fait. Pour de plus amples informations, veuillez consulter Définition du gestionnaire de fonction Lambda dans Node.js.

Lambda : exécution d'une version ou d'un alias de fonction imprévu

Problème : la version ou l'alias de la fonction n'est pas invoqué

Lorsque vous publiez de nouvelles fonctions Lambda dans la console ou que vous utilisez AWS SAM, la dernière version du code est représentée par. `$LATEST` Par défaut, les invocations qui ne spécifient pas de version ou d'alias ciblent automatiquement la version `$LATEST` du code de votre fonction.

Si vous utilisez des versions de fonction ou des alias spécifiques, il s'agit de versions publiées immuables d'une fonction en plus de `$LATEST`. Lors du dépannage de ces fonctions, déterminez d'abord si l'appelant a invoqué la version ou l'alias souhaité. Vous pouvez le faire en consultant vos journaux de fonction. La version de la fonction qui a été invoquée est toujours affichée dans la ligne de journal START :

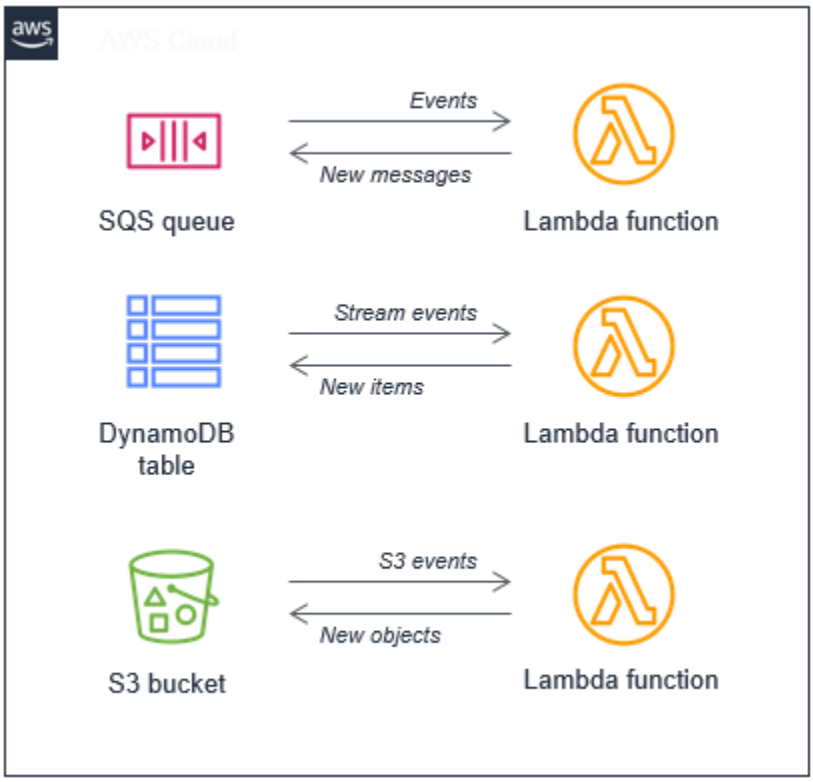


Lambda : détection de boucles infinies

Problème : modèles de boucles infinis liés aux fonctions Lambda

Il existe deux types de boucles infinies dans les fonctions Lambda. Le premier se trouve dans la fonction elle-même, à cause d'une boucle qui ne sort jamais. L'invocation prend fin uniquement lorsque la fonction expire. Vous pouvez identifier ces boucles en surveillant les expirations de délai, puis en corrigeant le comportement de bouclage.

Le second type de boucle se situe entre les fonctions Lambda et les autres AWS ressources. Elles se produisent lorsqu'un événement provenant d'une ressource telle qu'un compartiment S3 appelle une fonction Lambda, qui interagit ensuite avec la même ressource source pour déclencher un autre événement. Cela invoque à nouveau la fonction, qui crée une autre interaction avec le même compartiment S3, et ainsi de suite. Ces types de boucles peuvent être provoqués par différentes sources d' AWS événements, notamment les files d'attente Amazon SQS et les tables DynamoDB. Vous pouvez utiliser la détection de boucle récursive pour identifier ces modèles.



Vous pouvez éviter ces boucles en veillant à ce que les fonctions Lambda écrivent dans des ressources différentes de celles qui les consomment. Si vous devez republier les données sur la ressource consommatrice, assurez-vous que les nouvelles données ne déclenchent pas le même événement. Vous pouvez également utiliser le filtrage des événements. Par exemple, voici deux solutions proposées pour les boucles infinies avec les ressources S3 et DynamoDB :

- Si vous réécrivez dans le même compartiment S3, utilisez un préfixe ou un suffixe différent de celui du déclencheur d'événement.
- Si vous écrivez des éléments dans la même table DynamoDB, incluez un attribut sur lequel une fonction Lambda consommatrice peut filtrer. Si Lambda trouve l'attribut, il n'en résultera aucune autre invocation.

Général : indisponibilité du service en aval

Problème : les services en aval sur lesquels repose votre fonction Lambda ne sont pas disponibles

Pour les fonctions Lambda qui font appel à des points de terminaison tiers ou à d'autres ressources en aval, assurez-vous qu'elles sont capables de gérer les erreurs de service et les délais d'expiration.

Ces ressources en aval peuvent avoir des temps de réponse variables ou devenir indisponibles en raison d'interruptions de service. Selon l'implémentation, ces erreurs en aval peuvent apparaître sous forme de délais d'expiration ou d'exceptions Lambda, si la réponse d'erreur du service n'est pas traitée dans le code de la fonction.

Chaque fois qu'une fonction dépend d'un service en aval, tel qu'un appel d'API, implémentez une gestion des erreurs et une logique de nouvelle tentative appropriées. Pour les services critiques, la fonction Lambda doit publier des métriques ou des journaux sur. CloudWatch Par exemple, si une API de paiement tierce n'est plus disponible, votre fonction Lambda peut journaliser ces informations. Vous pouvez ensuite configurer des CloudWatch alarmes pour envoyer des notifications relatives à ces erreurs.

Étant donné que Lambda peut rapidement se mettre à l'échelle, les services en aval avec serveur peuvent avoir du mal à gérer les pics de trafic. Il existe trois approches courantes pour gérer ce problème :

- Mise en cache : envisagez de mettre en cache le résultat des valeurs renvoyées par des services tiers si elles ne changent pas fréquemment. Vous pouvez stocker ces valeurs dans une variable globale, dans votre fonction ou dans un autre service. Par exemple, les résultats d'une requête de liste de produits provenant d'une instance Amazon RDS peuvent être enregistrés pendant un certain temps dans la fonction afin d'éviter les requêtes redondantes.
- Mise en file d'attente : lors de l'enregistrement ou de la mise à jour de données, ajoutez une file d'attente Amazon SQS entre la fonction Lambda et la ressource. La file d'attente conserve les données de manière durable pendant que le service en aval traite les messages.
- Proxies : lorsque des connexions de longue durée sont généralement utilisées, comme pour les instances Amazon RDS, utilisez une couche proxy pour regrouper et réutiliser ces connexions. Pour les bases de données relationnelles, le Proxy Amazon RDS est un service conçu pour améliorer la capacité de mise à l'échelle et la résilience des applications basées sur Lambda.

AWS SDK : versions et mises à jour

Problème : le AWS SDK inclus dans le moteur d'exécution n'est pas la dernière version

Problème : le AWS SDK inclus dans le moteur d'exécution est automatiquement mis à jour

Les environnements d'exécution pour les langages interprétés incluent une version du AWS SDK. Lambda met régulièrement à jour ces environnements d'exécution pour utiliser la dernière version du

SDK. Pour trouver la version du SDK incluse dans votre environnement d'exécution, consultez les sections suivantes :

- Versions du SDK incluses dans l'environnement d'exécution (Node.js)
- Versions du SDK incluses dans l'environnement d'exécution (Python)
- Versions du SDK incluses dans l'environnement d'exécution (Ruby)

Pour utiliser une version plus récente du AWS SDK ou pour verrouiller vos fonctions sur une version spécifique, vous pouvez regrouper la bibliothèque avec votre code de fonction ou créer une couche Lambda. Pour plus d'informations sur la création d'un package de déploiement avec des dépendances, consultez les rubriques suivantes :

Node.js

Déployer des fonctions Lambda en Node.js avec des archives de fichiers .zip

Python

Travailler avec des archives de fichiers .zip pour les fonctions Lambda Python

Ruby

Déployer des fonctions Lambda en Ruby avec des archives de fichiers .zip

Java

Déployer des fonctions Lambda en Java avec des archives de fichiers .zip ou JAR

Go

Déployer des fonctions Lambda Go avec des archives de fichiers .zip

C#

Créez et déployez des fonctions Lambda C# à l'aide des archives de fichiers .zip

PowerShell

Déployer des fonctions Lambda PowerShell avec des archives de fichiers .zip

Python : les bibliothèques se chargent de manière incorrecte

Problème : (Python) certaines bibliothèques ne se chargent pas correctement à partir du package de déploiement

Les bibliothèques avec des modules d'extension écrits en C ou C++ doivent être compilées dans un environnement avec la même architecture de processeur que Lambda (Amazon Linux). Pour de plus amples informations, veuillez consulter Travailler avec des archives de fichiers .zip pour les fonctions Lambda Python.

Java : votre fonction met plus de temps à traiter les événements après la mise à jour de Java 11 vers Java 17

Probème : (Java) votre fonction met plus de temps à traiter les événements après la mise à jour de Java 11 vers Java 17

Réglez votre compilateur à l'aide du paramètre `JAVA_TOOL_OPTIONS`. Les environnements d'exécution Lambda pour Java 17 et versions ultérieures modifient les options du compilateur par défaut. Cette modification améliore les temps de démarrage à froid pour les fonctions de courte durée, mais le comportement précédent est mieux adapté aux fonctions de longue durée gourmandes en calcul. Définissez `JAVA_TOOL_OPTIONS` sur `-XX:-TieredCompilation` pour revenir au comportement de Java 11. Pour plus d'informations sur le paramètre `JAVA_TOOL_OPTIONS`, consultez the section called "Présentation de la variable d'environnement `JAVA_TOOL_OPTIONS`".

Kafka : Problèmes de gestion des erreurs et de configuration des nouvelles tentatives

Problème : le mappage de la source d'événements Kafka ne parvient pas à configurer les paramètres de nouvelle tentative ou les destinations en cas d'échec

Les configurations de nouvelle tentative de Kafka et les destinations en cas d'échec ne sont disponibles que pour les mappages de sources d'événements avec le mode provisionné activé. Assurez-vous d'avoir configuré votre configuration `ProvisionedPollerConfig` avant `MinimumPollers` de tenter de définir des configurations de nouvelle tentative.

Erreurs de configuration courantes :

- Nombre infini de tentatives avec un lot divisé en deux : vous ne pouvez pas l'activer `BisectBatchOnFunctionError` lorsque `MaximumRetryAttempts` ce paramètre est défini sur -1 (infini). Définissez une limite de tentatives limitée ou désactivez le traitement par lots en deux.

- Récursivité sur le même sujet — Le sujet de destination de Kafka en cas d'échec ne peut être identique à aucun de vos sujets source. Choisissez un autre nom de sujet pour votre sujet lettre morte.
- Format de destination Kafka non valide : utilisez le `kafka://<topic-name>` format lorsque vous spécifiez un sujet Kafka comme destination en cas d'échec.
- kafka : problèmes WriteData d'autorisation — Assurez-vous que votre rôle d'exécution dispose `kafka-cluster:WriteData` des autorisations pour le sujet de destination. Le sujet n'existe pas, les exceptions de délai d'expiration ou les problèmes de limitation de l'API d'écriture peuvent nécessiter une augmentation des limites du compte.

Résoudre les problèmes de mappage des sources d’événements dans Lambda

Dans Lambda, les problèmes liés au mappage des sources d’événements peuvent être plus complexes, car ils impliquent le débogage sur plusieurs services. De plus, le comportement de la source d’événements peut différer en fonction de la source d’événements exacte utilisée. Cette section répertorie les problèmes courants liés aux mappages des sources d’événements et fournit des conseils sur la manière de les identifier et de les résoudre.

> **Note**
>
> Cette section utilise une source d’événements Amazon SQS à titre d’illustration, mais les principes s’appliquent aux autres mappages des sources d’événements qui mettent en file d’attente des messages pour les fonctions Lambda.

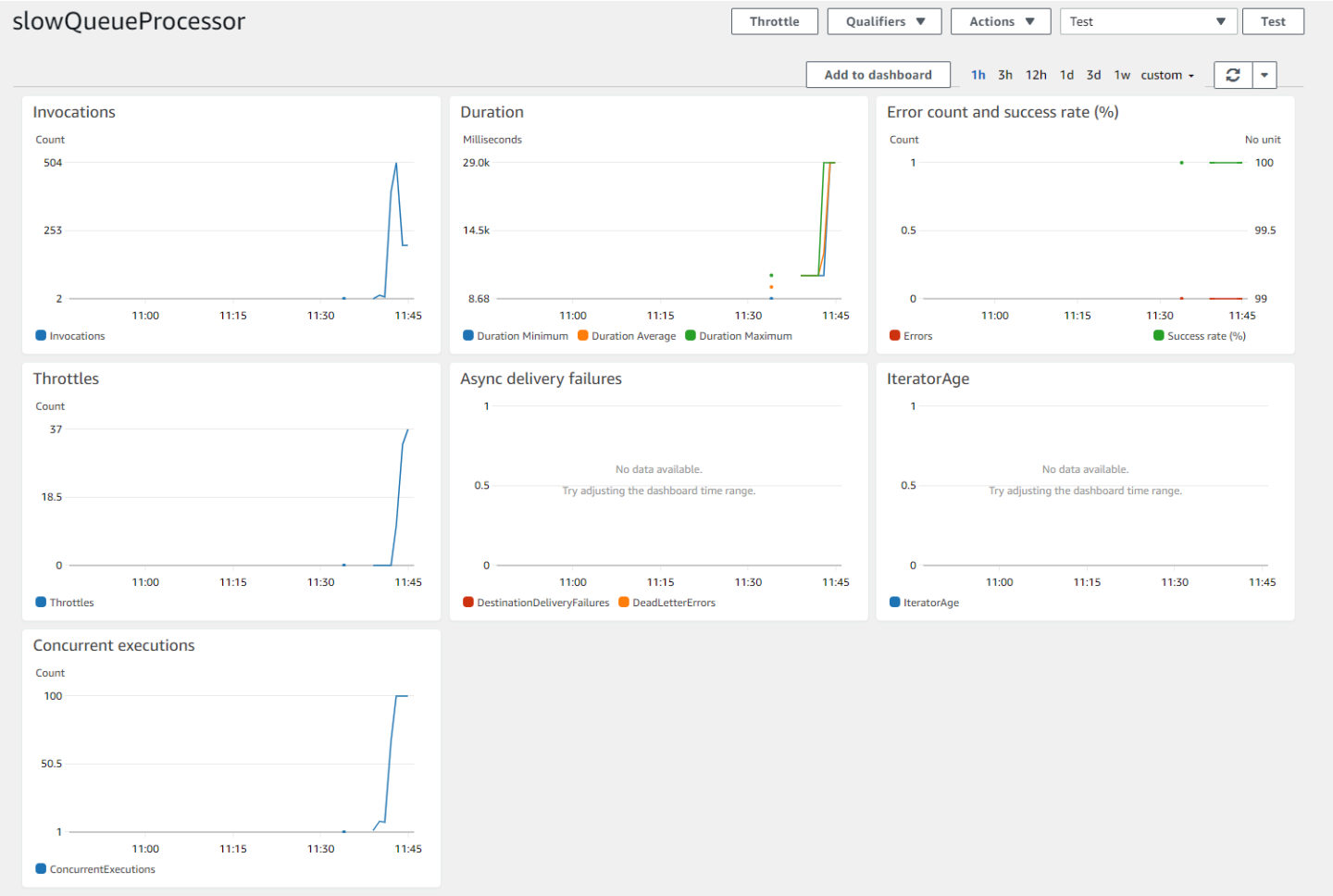
Identification et gestion de la limitation

Dans Lambda, la limitation se produit lorsque vous atteignez la limite de simultanéité de votre fonction ou de votre compte. Prenons l’exemple suivant, où une fonction Lambda lit les messages d’une file d’attente Amazon SQS. Cette fonction Lambda simule des appels de 30 secondes et a une taille de lot de 1. Cela signifie que la fonction ne traite qu’un seul message toutes les 30 secondes :

```
const doWork = (ms) => new Promise(resolve => setTimeout(resolve, ms))

exports.handler = async (event) => {
    await doWork(30000)
```

```
}
```

Avec une durée d'invocation aussi longue, les messages commencent à arriver dans la file d'attente plus rapidement qu'ils sont traités. Si la simultanéité non réservée de votre compte est de 100, Lambda augmente verticalement jusqu'à 100 exécutions simultanées, puis une limitation se produit. Vous pouvez voir ce schéma dans les métriques CloudWatch relatives à la fonction :



Les métriques CloudWatch relatives à la fonction n'indiquent aucune erreur, mais le graphique Exécutions simultanées indique que la simultanéité maximale de 100 est atteinte. Par conséquent, le graphique Limitations montre la limitation en place.

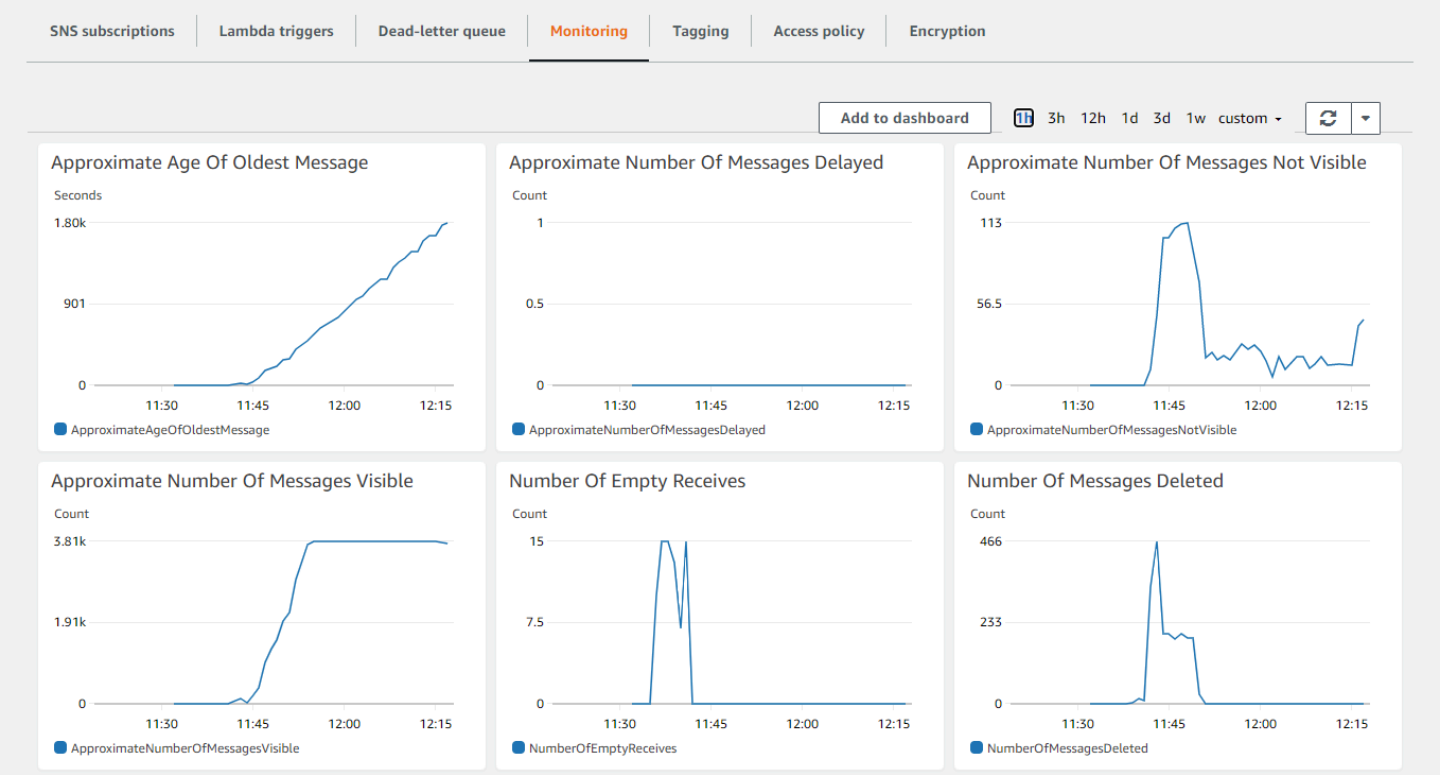
Vous pouvez détecter la limitation avec les alarmes CloudWatch et en définissant une alarme chaque fois que la métrique de limitation d'une fonction est supérieure à 0. Après avoir identifié le problème de limitation, plusieurs options s'offrent à vous pour le résoudre :

- Demander une augmentation de la simultanéité auprès d'AWS Support dans cette région.

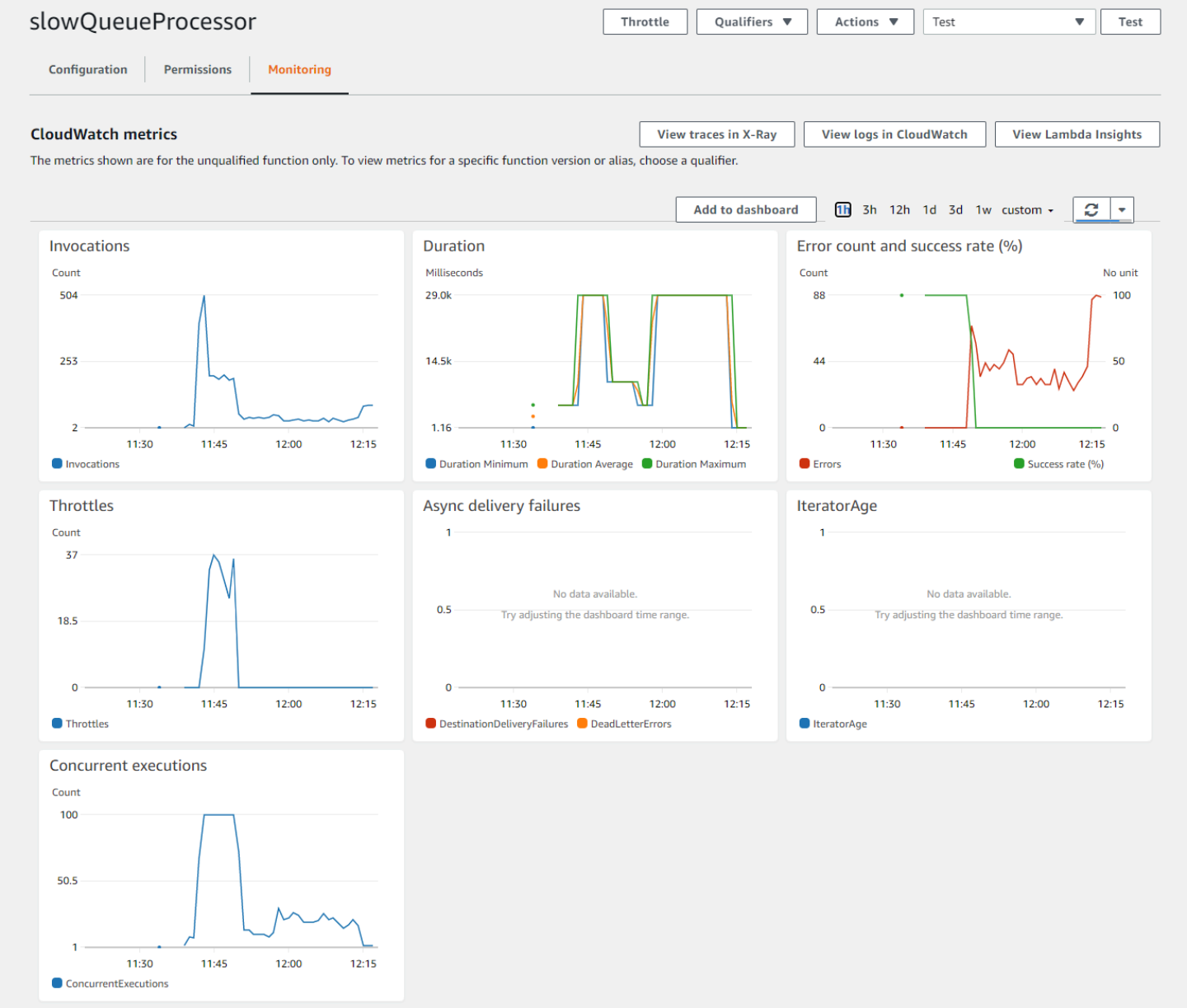
- Identifier les problèmes de performance de la fonction afin d'améliorer la vitesse de traitement et donc le débit.
- Augmenter la taille du lot de la fonction, de sorte que davantage de messages soient traités à chaque invocation.

Erreurs dans la fonction de traitement

Si la fonction de traitement génère des erreurs, Lambda renvoie les messages dans la file d'attente SQS. Lambda empêche la mise à l'échelle de votre fonction afin d'éviter les erreurs à grande échelle. Les métriques SQS suivantes dans CloudWatch indiquent qu'il existe un problème de traitement des files d'attente :

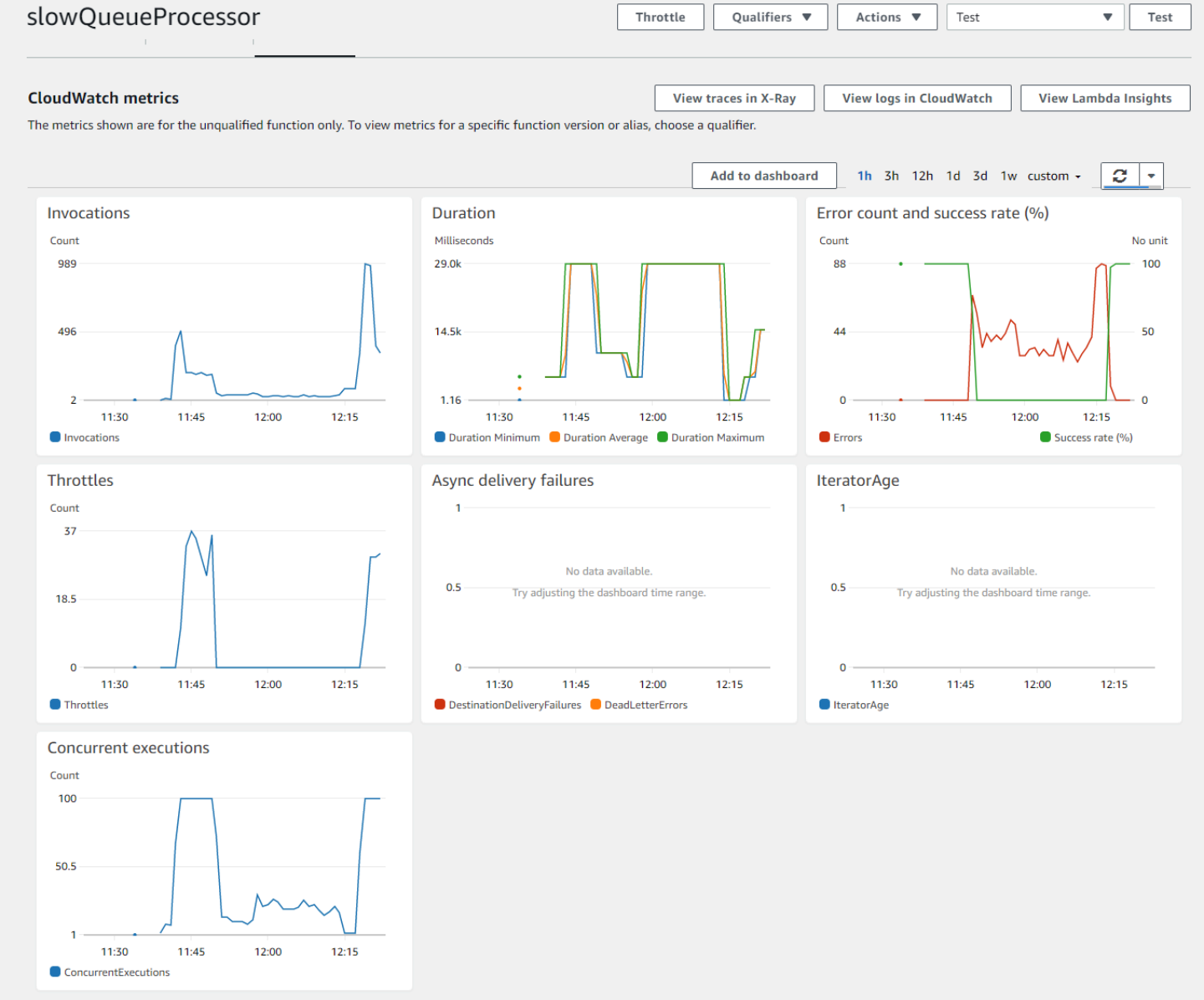


En particulier, l'âge du message le plus ancien et le nombre de messages visibles augmentent, alors qu'aucun message n'est supprimé. La file d'attente continue de croître, mais les messages ne sont pas traités. Les métriques CloudWatch relatives à la fonction Lambda de traitement indiquent également l'existence d'un problème :



La métrique du Nombre d’erreurs est différente de zéro et augmente, tandis que les Exécutions simultanées ont diminué et que la limitation a cessé. Cela montre que Lambda a cessé de se mettre à l’échelle votre fonction en raison d’erreurs. Les journaux CloudWatch de cette fonction fournissent des informations détaillées sur le type d’erreur.

Vous pouvez résoudre ce problème en identifiant la fonction à l’origine de l’erreur, puis en recherchant et en résolvant l’erreur. Après avoir corrigé l'erreur et déployé le nouveau code de fonction, les métriques CloudWatch devraient indiquer que le traitement a repris :



Ici, la métrique Nombre d'erreurs tombe à zéro et la métrique Taux de réussite revient à 100 %. Lambda recommence à augmenter verticalement la fonction, comme indiqué dans le graphique Exécutions simultanées.

Identification et gestion de la contre-pression

Si un producteur d'événements génère régulièrement des messages pour une file d'attente SQS plus rapidement qu'une fonction Lambda peut le gérer, une contre-pression se produit. Dans ce cas, la surveillance SQS doit indiquer que l'âge du message le plus ancien augmente de façon linéaire, ainsi que le nombre approximatif de messages visibles. Vous pouvez détecter la contre-pression dans les files d'attente à l'aide des alarmes CloudWatch.

Les étapes à suivre pour remédier à la contre-pression dépendent de votre charge de travail. Si l'objectif principal est d'augmenter la capacité de traitement et le débit par la fonction Lambda, plus options se présentent à vous :

- Demander une augmentation de la simultanéité dans la région concernée auprès d'AWS Support.
- Augmenter la taille du lot de la fonction, de sorte que davantage de messages soient traités à chaque invocation.

Résolution des problèmes de réseaux dans Lambda

Par défaut, Lambda exécute vos fonctions dans un cloud privé virtuel (Virtual Private Cloud, VPC) interne disposant d'connectivité aux services AWS et à Internet. Pour accéder aux ressources réseau en local, vous pouvez configurer votre fonction afin qu'elle se connecte à un VPC de votre compte. Lorsque vous utilisez cette fonctionnalité, vous gérez l'accès Internet et la connectivité réseau de la fonction avec les ressources Amazon Virtual Private Cloud (Amazon VPC).

Les erreurs de connectivité réseau peuvent être dues à des problèmes liés à la configuration du routage de votre VPC, aux règles du groupe de sécurité, aux autorisations de rôle Gestion des identités et des accès AWS (IAM) ou à la traduction d'adresses réseau (NAT), ou à la disponibilité de ressources telles que les adresses IP ou les interfaces réseau. En fonction du problème, vous pouvez voir une erreur spécifique ou un délai d'attente si une requête ne peut pas atteindre sa destination.

Rubriques

- VPC : la fonction perd l'accès à Internet ou expire
- VPC : échec intermittent de la connexion TCP ou UDP
- VPC : la fonction doit être accessible Services AWS sans utiliser Internet
- VPC : limite d'interface réseau Elastic atteinte
- EC2: interface réseau élastique de type « lambda »
- DNS : échec de la connexion aux hôtes avec UNKNOWNHOSTEXCEPTION

VPC : la fonction perd l'accès à Internet ou expire

Problème : votre fonction Lambda perd l'accès Internet après la connexion à un VPC.

Erreur : Error: connect ETIMEDOUT 176.32.98.189:443

Erreur : Error: Task timed out after 10.00 seconds

Erreur ReadTimeoutError : Le délai de lecture a expiré. (délai de lecture = 15)

Lorsque vous connectez une fonction à un VPC, toutes les demandes sortantes passent par le VPC. Pour vous connecter à Internet, configurez votre VPC pour envoyer le trafic sortant depuis le sous-réseau de la fonction vers une passerelle NAT dans un sous-réseau public. Pour obtenir plus d'informations et des exemples de configurations VPC, consultez the section called "Accès Internet pour les fonctions VPC".

Si certaines de vos connexions TCP sont interrompues, consultez the section called "VPC : échec intermittent de la connexion TCP ou UDP" si votre sous-réseau utilise une liste de contrôle d'accès (NACL). Dans le cas contraire, cela est probablement dû à la fragmentation des paquets. Les fonctions Lambda ne peuvent pas traiter les requêtes TCP fragmentées entrantes, car Lambda ne prend pas en charge la fragmentation IP pour TCP ou ICMP.

VPC : échec intermittent de la connexion TCP ou UDP

> **Note**
>
> Ce problème ne s'applique que si votre sous-réseau utilise une liste de contrôle d'accès (ACL) réseau. Le réseau ACLs n'est pas nécessaire pour que Lambda se connecte à vos sous-réseaux.

Problème : Lambda perd par intermittence la connexion à vos sous-réseaux VPC, pour lesquels vous avez configuré une liste de contrôle d'accès (ACL) réseau.

Pour les fonctions Lambda activées par VPC, AWS crée un hyperplan ENIs dans le compte du client et utilise des ports éphémères pour connecter Lambda `1024 65535` au VPC du client. Si vous utilisez le réseau ACLs dans le sous-réseau cible, vous devez autoriser la plage de ports à la fois `1024 65535` pour TCP et UDP. Ne pas autoriser cette plage complète de ports peut entraîner des défaillances de connexion intermittentes.

VPC : la fonction doit être accessible Services AWS sans utiliser Internet

Problème : Votre fonction Lambda doit être accessible Services AWS sans utiliser Internet.

Pour connecter une fonction Services AWS depuis un sous-réseau privé sans accès à Internet, utilisez des points de terminaison VPC.

VPC : limite d’interface réseau Elastic atteinte

Erreur ENILimit ReachedException : La limite de l'interface Elastic network a été atteinte pour le VPC de la fonction.

Lorsque vous connectez une fonction Lambda à un VPC, Lambda crée une interface réseau Elastic pour chaque combinaison de sous-réseau et de groupe de sécurité attaché à cette fonction. Le quota de service par défaut est de 250 interfaces réseau par VPC. Pour demander une augmentation d’un quota, vous pouvez utiliser la console Service Quotas.

EC2: interface réseau élastique de type « lambda »

Code d'erreur : client. OperationNotPermitted

Message d’erreur : le groupe de sécurité ne peut pas être modifié pour ce type d’interface

Vous recevrez cette erreur si vous tentez de modifier une Interface réseau Elastic (ENI) gérée par Lambda. Le `ModifyNetworkInterfaceAttribute` n’est pas inclus dans l’API Lambda pour les opérations de mise à jour sur les interfaces réseau élastiques créées par Lambda.

DNS : échec de la connexion aux hôtes avec UNKNOWNHOSTEXCEPTION

Message d’erreur : UNKNOWNHOSTEXCEPTION

Les fonctions Lambda prennent en charge un maximum de 20 connexions TCP simultanées pour la résolution DNS. Votre fonction est peut-être en train d’épuiser cette limite. Les requêtes DNS les plus courantes sont effectuées via UDP. Si votre fonction établit uniquement des connexions DNS UDP, il est peu probable que ce soit votre problème. Cette erreur est souvent due à une mauvaise configuration ou à une infrastructure dégradée. Par conséquent, avant d’examiner votre trafic DNS en profondeur, vérifiez que votre infrastructure DNS est correctement configurée et saine et que votre fonction Lambda fait référence à un hôte spécifié dans le DNS.

Si vous diagnostiquez que votre problème est lié à la connexion TCP maximale, notez que vous ne pouvez pas demander d’augmentation de cette limite. Si votre fonction Lambda revient au DNS TCP en raison de données utiles DNS importantes, vérifiez que votre solution utilise des bibliothèques compatibles EDNS. Pour plus d’informations sur l’extension EDNS, consultez la RFC 6891. Si vos données utiles DNS dépassent régulièrement les tailles maximales EDNS, votre solution risque d’épuiser la limite DNS TCP.

Exemples d'applications Lambda

Le référentiel GitHub de ce guide comprend des exemples d'applications qui démontrent l'utilisation de divers langages et services AWS. Chaque exemple d'application comprend des scripts facilitant le déploiement et le nettoyage, ainsi que des ressources complémentaires.

Node.js

Exemples d'applications Lambda en Node.js

- blank-nodejs – Fonction Node.js qui montre l'utilisation de la journalisation, des variables d'environnement, du suivi de AWS X-Ray, des couches, des tests unitaires et du kit SDK AWS.
- nodejs-apig – Fonction avec un point de terminaison d'API public qui traite un événement d'API Gateway et renvoie une réponse HTTP.

Python

Exemples d'applications Lambda en Python

- blank-python – Fonction Python montrant l'utilisation de la journalisation, des variables d'environnement, du suivi AWS X-Ray, des couches, des tests unitaires et du kit SDK AWS.

Ruby

Exemples d'applications Lambda en Ruby

- blank-ruby – Fonction Ruby qui montre l'utilisation de la journalisation, des variables d'environnement, du suivi de AWS X-Ray, des couches, des tests unitaires et du kit SDK AWS.
- Exemples de code Ruby pour AWS Lambda – Exemples de code écrits en Ruby qui montrent comment interagir avec AWS Lambda.

Java

Exemples d'applications Lambda en Java

- example-java : fonction Java qui montre comment utiliser Lambda pour traiter les commandes. Cette fonction montre comment définir et désérialiser un objet d'événement d'entrée personnalisé, utiliser le SDK AWS et enregistrer les sorties.

- java-basic – Ensemble de fonctions Java minimales avec des tests unitaires et une configuration de journalisation variable.
- java events – Ensemble de fonctions Java contenant du code squelette permettant de gérer les événements de divers services tels qu'Amazon API Gateway, Amazon SQS et Amazon Kinesis. Ces fonctions utilisent la dernière version (3.0.0 et ultérieure) de la bibliothèque aws-lambda-java-events. Ces exemples ne nécessitent pas le kit SDK AWS comme dépendance.
- s3-java – Fonction Java qui traite les événements de notification d'Amazon S3 et utilise la bibliothèque de classes Java (JCL) pour créer des miniatures à partir de fichiers d'image chargés.
- layer-java : fonction Java qui illustre comment utiliser une couche Lambda pour empaqueter les dépendances séparément du code de votre fonction principale.

Exécution de cadres Java populaires sur Lambda

- spring-cloud-function-samples : un exemple de Spring qui montre comment utiliser le cadre Spring Cloud Function pour créer des fonctions AWS Lambda.
- Démo de l'application Spring Boot sans serveur : un exemple qui montre comment configurer une application Spring Boot typique dans une exécution Java gérée avec et sans SnapStart, ou en tant qu'image native GraalVM avec une exécution personnalisée.
- Démo de l'application Micronaut sans serveur : un exemple qui montre comment utiliser Micronaut dans une exécution Java gérée avec et sans SnapStart, ou en tant qu'image native GraalVM avec une exécution personnalisée. Pour en savoir plus, consultez les guides Micronaut/Lambda.
- Démo de l'application Quarkus sans serveur : un exemple qui montre comment utiliser Quarkus dans une exécution Java gérée avec et sans SnapStart, ou en tant qu'image native GraalVM avec une exécution personnalisée. Pour en savoir plus, consultez le guide Quarkus/Lambda et le guide Quarkus/SnapStart.

Go

Lambda fournit les exemples d'applications suivants pour l'environnement d'exécution Go :

Exemples d'applications Lambda en Go

- go-al2 – Une fonction Hello World qui renvoie l'adresse IP publique. Cette application utilise l'exécution personnalisée `provided.al2`.

- blank-go – Fonction Go montrant l'utilisation des bibliothèques Go de Lambda, de la journalisation, des variables d'environnement et du kit AWS SDK. Cette application utilise l'exécution `go1.x`.

C#

Exemples d'applications Lambda en C#

- blank-csharp – Fonction C# montrant l'utilisation des bibliothèques .NET de Lambda, la journalisation, les variables d'environnement, le suivi AWS X-Ray, les tests unitaires et le kit AWS SDK.
- blank-csharp-with-layer – Fonction C# qui utilise l'interface de ligne de commande .NET pour créer une couche permettant d'empaqueter les dépendances de la fonction.
- ec2-spot – Fonction qui gère les demandes d'instance Spot dans Amazon EC2.

PowerShell

Lambda fournit les exemples d'applications suivants pour PowerShell :

- blank-powershell – Fonction PowerShell qui montre l'utilisation de la journalisation, des variables d'environnement et du kit SDK AWS.

Pour déployer un exemple d'application, suivez les instructions de son fichier README.

Utilisation de Lambda avec un SDK AWS

AWS des kits de développement logiciel (SDKs) sont disponibles pour de nombreux langages de programmation courants. Chaque kit SDK fournit une API, des exemples de code et de la documentation qui facilitent la création d'applications par les développeurs dans leur langage préféré.

Documentation SDK	Exemples de code
AWS SDK pour C++	AWS SDK pour C++ exemples de code
AWS CLI	AWS CLI exemples de code
AWS SDK pour Go	AWS SDK pour Go exemples de code
AWS SDK pour Java	AWS SDK pour Java exemples de code
AWS SDK pour JavaScript	AWS SDK pour JavaScript exemples de code
AWS SDK pour Kotlin	AWS SDK pour Kotlin exemples de code
AWS SDK pour .NET	AWS SDK pour .NET exemples de code
AWS SDK pour PHP	AWS SDK pour PHP exemples de code
Outils AWS pour PowerShell	Outils AWS pour PowerShell exemples de code
AWS SDK pour Python (Boto3)	AWS SDK pour Python (Boto3) exemples de code
AWS SDK pour Ruby	AWS SDK pour Ruby exemples de code
AWS SDK pour Rust	AWS SDK pour Rust exemples de code
AWS SDK pour SAP ABAP	AWS SDK pour SAP ABAP exemples de code
AWS SDK pour Swift	AWS SDK pour Swift exemples de code

Pour plus d'exemples spécifiques à Lambda, consultez Exemples de code pour Lambda utilisant AWS SDKs.

Exemple de disponibilité

Vous n’avez pas trouvé ce dont vous avez besoin ? Demandez un exemple de code en utilisant le lien Provide feedback (Fournir un commentaire) en bas de cette page.

Exemples de code pour Lambda utilisant AWS SDKs

Les exemples de code suivants montrent comment utiliser Lambda avec un kit de développement AWS logiciel (SDK).

Les principes de base sont des exemples de code qui vous montrent comment effectuer les opérations essentielles au sein d'un service.

Les actions sont des extraits de code de programmes plus larges et doivent être exécutées dans leur contexte. Alors que les actions vous indiquent comment appeler des fonctions de service individuelles, vous pouvez les voir en contexte dans leurs scénarios associés.

Les scénarios sont des exemples de code qui vous montrent comment accomplir des tâches spécifiques en appelant plusieurs fonctions au sein d'un même service ou combinés à d'autres Services AWS.

AWS les contributions communautaires sont des exemples qui ont été créés et sont maintenus par plusieurs équipes AWS. Pour faire un commentaire, utilisez le mécanisme fourni dans les référentiels liés.

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit de développement logiciel (SDK).

Exemples de code

- Exemples de base pour l'utilisation de Lambda AWS SDKs
  - Hello Lambda
  - Apprenez les bases de Lambda avec un SDK AWS
  - Actions pour Lambda utilisant AWS SDKs
    - Utilisation de CreateAlias avec une CLI
    - Utilisation CreateFunction avec un AWS SDK ou une CLI
    - Utilisation de DeleteAlias avec une CLI
    - Utilisation DeleteFunction avec un AWS SDK ou une CLI
    - Utilisation de DeleteFunctionConcurrency avec une CLI
    - Utilisation de DeleteProvisionedConcurrencyConfig avec une CLI

- Utilisation de GetAccountSettings avec une CLI
- Utilisation de GetAlias avec une CLI
- Utilisation GetFunction avec un AWS SDK ou une CLI
- Utilisation de GetFunctionConcurrency avec une CLI
- Utilisation de GetFunctionConfiguration avec une CLI
- Utilisation de GetPolicy avec une CLI
- Utilisation de GetProvisionedConcurrencyConfig avec une CLI
- Utilisation Invoke avec un AWS SDK ou une CLI
- Utilisation ListFunctions avec un AWS SDK ou une CLI
- Utilisation de ListProvisionedConcurrencyConfigs avec une CLI
- Utilisation de ListTags avec une CLI
- Utilisation de ListVersionsByFunction avec une CLI
- Utilisation de PublishVersion avec une CLI
- Utilisation de PutFunctionConcurrency avec une CLI
- Utilisation de PutProvisionedConcurrencyConfig avec une CLI
- Utilisation de RemovePermission avec une CLI
- Utilisation de TagResource avec une CLI
- Utilisation de UntagResource avec une CLI
- Utilisation de UpdateAlias avec une CLI
- Utilisation UpdateFunctionCode avec un AWS SDK ou une CLI
- Utilisation UpdateFunctionConfiguration avec un AWS SDK ou une CLI

- Scénarios d'utilisation de Lambda AWS SDKs
  - Confirmez automatiquement les utilisateurs Amazon Cognito connus à l'aide d'une fonction Lambda à l'aide d'un SDK AWS
  - Migrez automatiquement les utilisateurs connus d'Amazon Cognito à l'aide d'une fonction Lambda à l'aide d'un SDK AWS
  - Créer une API REST API Gateway pour suivre les données de la COVID-19
  - Créer une API REST de bibliothèque de prêt
  - Créer une application de messagerie avec Step Functions
  - Création d’une application de gestion des ressources photographiques permettant aux utilisateurs de gérer les photos à l’aide d’étiquettes

- Créer une application de chat Websocket avec API Gateway
- Créez une application qui analyse les commentaires des clients et synthétise le son
- Invoquer une fonction Lambda à partir d'un navigateur
- Transformation des données pour votre application avec S3 Object Lambda
- Utiliser API Gateway pour invoquer une fonction Lambda
- Utiliser les fonctions Step Functions pour invoquer des fonctions Lambda
- Utilisent des événements planifiés pour invoquer une fonction Lambda
- Utiliser l'API Amazon Neptune pour développer une fonction Lambda qui interroge les données d'un graphe
- Rédigez des données d'activité personnalisées à l'aide d'une fonction Lambda après l'authentification de l'utilisateur Amazon Cognito à l'aide d'un SDK AWS

- Exemples sans serveur pour Lambda
  - Connexion à une base de données Amazon RDS dans une fonction Lambda
  - Invoquer une fonction Lambda à partir d'un déclencheur Kinesis
  - Invocation d'une fonction Lambda à partir d'un déclencheur DynamoDB
  - Invocation d'une fonction Lambda à partir d'un déclencheur Amazon DocumentDB
  - Invocation d'une fonction Lambda à partir d'un déclencheur Amazon MSK
  - Invoquer une fonction Lambda à partir d'un déclencheur Amazon S3
  - Invocation d'une fonction lambda à partir d'un déclencheur Amazon SNS
  - Invoquer une fonction Lambda à partir d'un déclencheur Amazon SQS
  - Signalement des échecs d'articles par lots pour les fonctions Lambda à l'aide d'un déclencheur Kinesis
  - Signalement des échecs d'articles par lots pour les fonctions Lambda à l'aide d'un déclencheur DynamoDB
  - Signalement des échecs d'articles par lots pour les fonctions Lambda à l'aide d'un déclencheur Amazon SQS
- AWS contributions de la communauté pour Lambda
  - Création et test d'une application sans serveur

Exemples de base pour l'utilisation de Lambda AWS SDKs

Les exemples de code suivants montrent comment utiliser les principes de base de AWS Lambda with AWS SDKs.

Exemples

- Hello Lambda
- Apprenez les bases de Lambda avec un SDK AWS
- Actions pour Lambda utilisant AWS SDKs
  - Utilisation de CreateAlias avec une CLI
  - Utilisation CreateFunction avec un AWS SDK ou une CLI
  - Utilisation de DeleteAlias avec une CLI
  - Utilisation DeleteFunction avec un AWS SDK ou une CLI
  - Utilisation de DeleteFunctionConcurrency avec une CLI
  - Utilisation de DeleteProvisionedConcurrencyConfig avec une CLI
  - Utilisation de GetAccountSettings avec une CLI
  - Utilisation de GetAlias avec une CLI
  - Utilisation GetFunction avec un AWS SDK ou une CLI
  - Utilisation de GetFunctionConcurrency avec une CLI
  - Utilisation de GetFunctionConfiguration avec une CLI
  - Utilisation de GetPolicy avec une CLI
  - Utilisation de GetProvisionedConcurrencyConfig avec une CLI
  - Utilisation Invoke avec un AWS SDK ou une CLI
  - Utilisation ListFunctions avec un AWS SDK ou une CLI
  - Utilisation de ListProvisionedConcurrencyConfigs avec une CLI
  - Utilisation de ListTags avec une CLI
  - Utilisation de ListVersionsByFunction avec une CLI
  - Utilisation de PublishVersion avec une CLI
  - Utilisation de PutFunctionConcurrency avec une CLI
  - Utilisation de PutProvisionedConcurrencyConfig avec une CLI
  - Utilisation de RemovePermission avec une CLI
  - Utilisation de TagResource avec une CLI

- Utilisation de UntagResource avec une CLI
- Utilisation de UpdateAlias avec une CLI
- Utilisation UpdateFunctionCode avec un AWS SDK ou une CLI
- Utilisation UpdateFunctionConfiguration avec un AWS SDK ou une CLI

Hello Lambda

Les exemples de code suivants montrent comment démarrer avec Lambda.

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
namespace LambdaActions;

using Amazon.Lambda;

public class HelloLambda
{
    static async Task Main(string[] args)
    {
        var lambdaClient = new AmazonLambdaClient();

        Console.WriteLine("Hello AWS Lambda");
        Console.WriteLine("Let's get started with AWS Lambda by listing your
 existing Lambda functions:");

        var response = await lambdaClient.ListFunctionsAsync();
        response.Functions.ForEach(function =>
        {

 Console.WriteLine($"{function.FunctionName}\t{function.Description}");
        });
    }
```

```
}
```

- Pour plus de détails sur l'API, reportez-vous ListFunctionsà la section Référence des AWS SDK pour .NET API.

C++

SDK pour C++

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

Code pour le CMake fichier CMake Lists.txt.

```
# Set the minimum required version of CMake for this project.
cmake_minimum_required(VERSION 3.13)

# Set the AWS service components used by this project.
set(SERVICE_COMPONENTS lambda)

# Set this project's name.
project("hello_lambda")

# Set the C++ standard to use to build this target.
# At least C++ 11 is required for the AWS SDK for C++.
set(CMAKE_CXX_STANDARD 11)

# Use the MSVC variable to determine if this is a Windows build.
set(WINDOWS_BUILD ${MSVC})

if (WINDOWS_BUILD) # Set the location where CMake can find the installed
 libraries for the AWS SDK.
    string(REPLACE ";" "/aws-cpp-sdk-all;" SYSTEM_MODULE_PATH
 "${CMAKE_SYSTEM_PREFIX_PATH}/aws-cpp-sdk-all")
    list(APPEND CMAKE_PREFIX_PATH ${SYSTEM_MODULE_PATH})
endif ()
```

```
# Find the AWS SDK for C++ package.
find_package(AWSSDK REQUIRED COMPONENTS ${SERVICE_COMPONENTS})

if (WINDOWS_BUILD AND AWSSDK_INSTALL_AS_SHARED_LIBS)
     # Copy relevant AWS SDK for C++ libraries into the current binary directory
 for running and debugging.

     # set(BIN_SUB_DIR "/Debug") # if you are building from the command line you
 may need to uncomment this
                                     # and set the proper subdirectory to the
 executables' location.

     AWSSDK_CPY_DYN_LIBS(SERVICE_COMPONENTS ""
 ${CMAKE_CURRENT_BINARY_DIR}${BIN_SUB_DIR})
endif ()

add_executable(${PROJECT_NAME}
        hello_lambda.cpp)

target_link_libraries(${PROJECT_NAME}
        ${AWSSDK_LINK_LIBRARIES})
```

Code pour le fichier source hello_lambda.cpp.

```
#include <aws/core/Aws.h>
#include <aws/lambda/LambdaClient.h>
#include <aws/lambda/model/ListFunctionsRequest.h>
#include <iostream>

/*
 *  A "Hello Lambda" starter application which initializes an AWS Lambda (Lambda)
 client and lists the Lambda functions.
 *
 *  main function
 *
 *  Usage: 'hello_lambda'
 *
 */

int main(int argc, char **argv) {
    Aws::SDKOptions options;
    // Optionally change the log level for debugging.
```

```
//   options.loggingOptions.logLevel = Utils::Logging::LogLevel::Debug;
    Aws::InitAPI(options); // Should only be called once.
    int result = 0;
    {
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";

        Aws::Lambda::LambdaClient lambdaClient(clientConfig);
        std::vector<Aws::String> functions;
        Aws::String marker; // Used for pagination.

        do {
            Aws::Lambda::Model::ListFunctionsRequest request;
            if (!marker.empty()) {
                request.SetMarker(marker);
            }

            Aws::Lambda::Model::ListFunctionsOutcome outcome =
 lambdaClient.ListFunctions(
                    request);

            if (outcome.IsSuccess()) {
                const Aws::Lambda::Model::ListFunctionsResult
 &listFunctionsResult = outcome.GetResult();
                std::cout << listFunctionsResult.GetFunctions().size()
                          << " lambda functions were retrieved." << std::endl;

                for (const Aws::Lambda::Model::FunctionConfiguration
 &functionConfiguration: listFunctionsResult.GetFunctions()) {
                    functions.push_back(functionConfiguration.GetFunctionName());
                    std::cout << functions.size() << "  "
                              << functionConfiguration.GetDescription() <<
 std::endl;
                    std::cout << "   "
                              <<
 Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(
                                      functionConfiguration.GetRuntime()) << ": "
                              << functionConfiguration.GetHandler()
                              << std::endl;
                }
                marker = listFunctionsResult.GetNextMarker();
            } else {
                std::cerr << "Error with Lambda::ListFunctions. "
```

```
                        << outcome.GetError().GetMessage()
                        << std::endl;
                result = 1;
                break;
            }
        } while (!marker.empty());
    }


    Aws::ShutdownAPI(options); // Should only be called once.
    return result;
}
```

- Pour plus de détails sur l'API, reportez-vous ListFunctionsà la section Référence des AWS SDK pour C++ API.

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
package main

import (
 "context"
 "fmt"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
)

// main uses the AWS SDK for Go (v2) to create an AWS Lambda client and list up
 to 10
```

```
// functions in your account.
// This example uses the default settings specified in your shared credentials
// and config files.
func main() {
 ctx := context.Background()
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  fmt.Println("Couldn't load default configuration. Have you set up your AWS
 account?")
  fmt.Println(err)
  return
 }
 lambdaClient := lambda.NewFromConfig(sdkConfig)

 maxItems := 10
 fmt.Printf("Let's list up to %v functions for your account.\n", maxItems)
 result, err := lambdaClient.ListFunctions(ctx, &lambda.ListFunctionsInput{
  MaxItems: aws.Int32(int32(maxItems)),
 })
 if err != nil {
  fmt.Printf("Couldn't list functions for your account. Here's why: %v\n", err)
  return
 }
 if len(result.Functions) == 0 {
  fmt.Println("You don't have any functions!")
 } else {
  for _, function := range result.Functions {
   fmt.Printf("\t%v\n", *function.FunctionName)
  }
 }
}
```

- Pour plus de détails sur l'API, reportez-vous ListFunctionsà la section Référence des AWS SDK pour Go API.

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    /**
     * Lists the AWS Lambda functions associated with the current AWS account.
     *
     * @param awsLambda an instance of the {@link LambdaClient} class, which is
used to interact with the AWS Lambda service
     *
     * @throws LambdaException if an error occurs while interacting with the AWS
Lambda service
     */
    public static void listFunctions(LambdaClient awsLambda) {
        try {
            ListFunctionsResponse functionResult = awsLambda.listFunctions();
            List<FunctionConfiguration> list = functionResult.functions();
            for (FunctionConfiguration config : list) {
                System.out.println("The function name is " +
config.functionName());
            }

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

- Pour plus de détails sur l'API, reportez-vous ListFunctionsà la section Référence des AWS SDK for Java 2.x API.

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
import { LambdaClient, paginateListFunctions } from "@aws-sdk/client-lambda";

const client = new LambdaClient({});

export const helloLambda = async () => {
  const paginator = paginateListFunctions({ client }, {});
  const functions = [];

  for await (const page of paginator) {
    const funcNames = page.Functions.map((f) => f.FunctionName);
    functions.push(...funcNames);
  }

  console.log("Functions:");
  console.log(functions.join("\n"));
  return functions;
};
```

- Pour plus de détails sur l'API, reportez-vous ListFunctionsà la section Référence des AWS SDK pour JavaScript API.

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
import boto3


def main():
    """
    List the Lambda functions in your AWS account.
    """
    # Create the Lambda client
    lambda_client = boto3.client("lambda")

    # Use the paginator to list the functions
    paginator = lambda_client.get_paginator("list_functions")
    response_iterator = paginator.paginate()

    print("Here are the Lambda functions in your account:")
    for page in response_iterator:
        for function in page["Functions"]:
            print(f"  {function['FunctionName']}")


if __name__ == "__main__":
    main()
```

- Pour plus de détails sur l'API, consultez ListFunctionsle AWS manuel de référence de l'API SDK for Python (Boto3).

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
require 'aws-sdk-lambda'
```

```
# Creates an AWS Lambda client using the default credentials and configuration
def lambda_client
  Aws::Lambda::Client.new
end

# Lists the Lambda functions in your AWS account, paginating the results if
 necessary
def list_lambda_functions
  lambda = lambda_client

  # Use a pagination iterator to list all functions
  functions = []
  lambda.list_functions.each_page do |page|
    functions.concat(page.functions)
  end

  # Print the name and ARN of each function
  functions.each do |function|
    puts "Function name: #{function.function_name}"
    puts "Function ARN: #{function.function_arn}"
    puts
  end

  puts "Total functions: #{functions.count}"
end

list_lambda_functions if __FILE__ == $PROGRAM_NAME
```

- Pour plus de détails sur l'API, reportez-vous ListFunctionsà la section Référence des AWS SDK pour Ruby API.

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Apprenez les bases de Lambda avec un SDK AWS

Les exemples de code suivants montrent comment :

- Créer un rôle IAM et une fonction Lambda, puis charger le code du gestionnaire.
- Invoquer la fonction avec un seul paramètre et obtenir des résultats.
- Mettre à jour le code de la fonction et configurer avec une variable d'environnement.
- Invoquer la fonction avec de nouveaux paramètres et obtenir des résultats. Afficher le journal d'exécution renvoyé.
- Répertorier les fonctions pour votre compte, puis nettoyer les ressources.

Pour plus d'informations, consultez Créer une fonction Lambda à l'aide de la console.

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

Créez des méthodes qui exécutent des actions Lambda.

```
namespace LambdaActions;

using Amazon.Lambda;
using Amazon.Lambda.Model;

/// <summary>
/// A class that implements AWS Lambda methods.
/// </summary>
public class LambdaWrapper
{
    private readonly IAmazonLambda _lambdaService;

    /// <summary>
    /// Constructor for the LambdaWrapper class.
    /// </summary>
    /// <param name="lambdaService">An initialized Lambda service client.</param>
    public LambdaWrapper(IAmazonLambda lambdaService)
    {
        _lambdaService = lambdaService;
```

```
    }

    /// <summary>
    /// Creates a new Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the function.</param>
    /// <param name="s3Bucket">The Amazon Simple Storage Service (Amazon S3)
    /// bucket where the zip file containing the code is located.</param>
    /// <param name="s3Key">The Amazon S3 key of the zip file.</param>
    /// <param name="role">The Amazon Resource Name (ARN) of a role with the
    /// appropriate Lambda permissions.</param>
    /// <param name="handler">The name of the handler function.</param>
    /// <returns>The Amazon Resource Name (ARN) of the newly created
    /// Lambda function.</returns>
    public async Task<string> CreateLambdaFunctionAsync(
        string functionName,
        string s3Bucket,
        string s3Key,
        string role,
        string handler)
    {
        // Defines the location for the function code.
        // S3Bucket - The S3 bucket where the file containing
        //            the source code is stored.
        // S3Key    - The name of the file containing the code.
        var functionCode = new FunctionCode
        {
            S3Bucket = s3Bucket,
            S3Key = s3Key,
        };

        var createFunctionRequest = new CreateFunctionRequest
        {
            FunctionName = functionName,
            Description = "Created by the Lambda .NET API",
            Code = functionCode,
            Handler = handler,
            Runtime = Runtime.Dotnet6,
            Role = role,
        };

        var reponse = await
_lambdaService.CreateFunctionAsync(createFunctionRequest);
        return reponse.FunctionArn;
```

```
    }


    /// <summary>
    /// Delete an AWS Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// delete.</param>
    /// <returns>A Boolean value that indicates the success of the action.</
returns>
    public async Task<bool> DeleteFunctionAsync(string functionName)
    {
        var request = new DeleteFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.DeleteFunctionAsync(request);

        // A return value of NoContent means that the request was processed.
        // In this case, the function was deleted, and the return value
        // is intentionally blank.
        return response.HttpStatusCode == System.Net.HttpStatusCode.NoContent;
    }


    /// <summary>
    /// Gets information about a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function for
    /// which to retrieve information.</param>
    /// <returns>Async Task.</returns>
    public async Task<FunctionConfiguration> GetFunctionAsync(string
 functionName)
    {
        var functionRequest = new GetFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.GetFunctionAsync(functionRequest);
        return response.Configuration;
    }
```

```
    /// <summary>
    /// Invoke a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// invoke.</param
    /// <param name="parameters">The parameter values that will be passed to the
function.</param>
    /// <returns>A System Threading Task.</returns>
    public async Task<string> InvokeFunctionAsync(
        string functionName,
        string parameters)
    {
        var payload = parameters;
        var request = new InvokeRequest
        {
            FunctionName = functionName,
            Payload = payload,
        };

        var response = await _lambdaService.InvokeAsync(request);
        MemoryStream stream = response.Payload;
        string returnValue =
System.Text.Encoding.UTF8.GetString(stream.ToArray());
        return returnValue;
    }


    /// <summary>
    /// Get a list of Lambda functions.
    /// </summary>
    /// <returns>A list of FunctionConfiguration objects.</returns>
    public async Task<List<FunctionConfiguration>> ListFunctionsAsync()
    {
        var functionList = new List<FunctionConfiguration>();

        var functionPaginator =
            _lambdaService.Paginators.ListFunctions(new ListFunctionsRequest());
        await foreach (var function in functionPaginator.Functions)
        {
            functionList.Add(function);
        }

        return functionList;
```

```
    }


    /// <summary>
    /// Update an existing Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to update.</
param>
    /// <param name="bucketName">The bucket where the zip file containing
    /// the Lambda function code is stored.</param>
    /// <param name="key">The key name of the source code file.</param>
    /// <returns>Async Task.</returns>
    public async Task UpdateFunctionCodeAsync(
        string functionName,
        string bucketName,
        string key)
    {
        var functionCodeRequest = new UpdateFunctionCodeRequest
        {
            FunctionName = functionName,
            Publish = true,
            S3Bucket = bucketName,
            S3Key = key,
        };

        var response = await
 _lambdaService.UpdateFunctionCodeAsync(functionCodeRequest);
        Console.WriteLine($"The Function was last modified at
 {response.LastModified}.");
    }


    /// <summary>
    /// Update the code of a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the function to update.</param>
    /// <param name="functionHandler">The code that performs the function's
 actions.</param>
    /// <param name="environmentVariables">A dictionary of environment
 variables.</param>
    /// <returns>A Boolean value indicating the success of the action.</returns>
    public async Task<bool> UpdateFunctionConfigurationAsync(
        string functionName,
        string functionHandler,
```

```
        Dictionary<string, string> environmentVariables)
    {
        var request = new UpdateFunctionConfigurationRequest
        {
            Handler = functionHandler,
            FunctionName = functionName,
            Environment = new Amazon.Lambda.Model.Environment { Variables =
 environmentVariables },
        };

        var response = await
 _lambdaService.UpdateFunctionConfigurationAsync(request);

        Console.WriteLine(response.LastModified);

        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }


}
```

Créer une fonction qui exécute le scénario.

```
global using System.Threading.Tasks;
global using Amazon.IdentityManagement;
global using Amazon.Lambda;
global using LambdaActions;
global using LambdaScenarioCommon;
global using Microsoft.Extensions.DependencyInjection;
global using Microsoft.Extensions.Hosting;
global using Microsoft.Extensions.Logging;
global using Microsoft.Extensions.Logging.Console;
global using Microsoft.Extensions.Logging.Debug;


using Amazon.Lambda.Model;
using Microsoft.Extensions.Configuration;

namespace LambdaBasics;

public class LambdaBasics
```

```
{
    private static ILogger logger = null!;

    static async Task Main(string[] args)
    {
        // Set up dependency injection for the Amazon service.
        using var host = Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logging =>
                logging.AddFilter("System", LogLevel.Debug)
                    .AddFilter<DebugLoggerProvider>("Microsoft",
 LogLevel.Information)
                    .AddFilter<ConsoleLoggerProvider>("Microsoft",
 LogLevel.Trace))
            .ConfigureServices((_, services) =>
            services.AddAWSService<IAmazonLambda>()
            .AddAWSService<IAmazonIdentityManagementService>()
            .AddTransient<LambdaWrapper>()
            .AddTransient<LambdaRoleWrapper>()
            .AddTransient<UIWrapper>()
        )
        .Build();

        var configuration = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("settings.json") // Load test settings from .json file.
            .AddJsonFile("settings.local.json",
            true) // Optionally load local settings.
        .Build();


        logger = LoggerFactory.Create(builder => { builder.AddConsole(); })
            .CreateLogger<LambdaBasics>();

        var lambdaWrapper = host.Services.GetRequiredService<LambdaWrapper>();
        var lambdaRoleWrapper =
 host.Services.GetRequiredService<LambdaRoleWrapper>();
        var uiWrapper = host.Services.GetRequiredService<UIWrapper>();

        string functionName = configuration["FunctionName"]!;
        string roleName = configuration["RoleName"]!;
        string policyDocument = "{" +
            " \"Version\": \"2012-10-17\"," +
            " \"Statement\": [ " +
            "    {" +
```

```
            "        \"Effect\": \"Allow\"," +
            "        \"Principal\": {" +
            "            \"Service\": \"lambda.amazonaws.com\" " +
            "    }," +
            "        \"Action\": \"sts:AssumeRole\" " +
            "    }" +
            "]" +
        "}";

        var incrementHandler = configuration["IncrementHandler"];
        var calculatorHandler = configuration["CalculatorHandler"];
        var bucketName = configuration["BucketName"];
        var incrementKey = configuration["IncrementKey"];
        var calculatorKey = configuration["CalculatorKey"];
        var policyArn = configuration["PolicyArn"];

        uiWrapper.DisplayLambdaBasicsOverview();

        // Create the policy to use with the AWS Lambda functions and then attach
the
        // policy to a new role.
        var roleArn = await lambdaRoleWrapper.CreateLambdaRoleAsync(roleName,
policyDocument);

        Console.WriteLine("Waiting for role to become active.");
        uiWrapper.WaitABit(15, "Wait until the role is active before trying to
use it.");

        // Attach the appropriate AWS Identity and Access Management (IAM) role
policy to the new role.
        var success = await
lambdaRoleWrapper.AttachLambdaRolePolicyAsync(policyArn, roleName);
        uiWrapper.WaitABit(10, "Allow time for the IAM policy to be attached to
the role.");

        // Create the Lambda function using a zip file stored in an Amazon Simple
Storage Service
        // (Amazon S3) bucket.
        uiWrapper.DisplayTitle("Create Lambda Function");
        Console.WriteLine($"Creating the AWS Lambda function: {functionName}.");
        var lambdaArn = await lambdaWrapper.CreateLambdaFunctionAsync(
            functionName,
            bucketName,
            incrementKey,
```

```
            roleArn,
            incrementHandler);

        Console.WriteLine("Waiting for the new function to be available.");
        Console.WriteLine($"The AWS Lambda ARN is {lambdaArn}");

        // Get the Lambda function.
        Console.WriteLine($"Getting the {functionName} AWS Lambda function.");
        FunctionConfiguration config;
        do
        {
            config = await lambdaWrapper.GetFunctionAsync(functionName);
            Console.Write(".");
        }
        while (config.State != State.Active);

        Console.WriteLine($"\nThe function, {functionName} has been created.");
        Console.WriteLine($"The runtime of this Lambda function is
{config.Runtime}.");

        uiWrapper.PressEnter();

        // List the Lambda functions.
        uiWrapper.DisplayTitle("Listing all Lambda functions.");
        var functions = await lambdaWrapper.ListFunctionsAsync();
        DisplayFunctionList(functions);

        uiWrapper.DisplayTitle("Invoke increment function");
        Console.WriteLine("Now that it has been created, invoke the Lambda
increment function.");
        string? value;
        do
        {
            Console.Write("Enter a value to increment: ");
            value = Console.ReadLine();
        }
        while (string.IsNullOrEmpty(value));

        string functionParameters = "{" +
            "\"action\": \"increment\", " +
            "\"x\": \"" + value + "\"" +
        "}";
        var answer = await lambdaWrapper.InvokeFunctionAsync(functionName,
functionParameters);
```

```
        Console.WriteLine($"{value} + 1 = {answer}.");

        uiWrapper.DisplayTitle("Update function");
        Console.WriteLine("Now update the Lambda function code.");
        await lambdaWrapper.UpdateFunctionCodeAsync(functionName, bucketName,
calculatorKey);

        do
        {
            config = await lambdaWrapper.GetFunctionAsync(functionName);
            Console.Write(".");
        }
        while (config.LastUpdateStatus == LastUpdateStatus.InProgress);

        await lambdaWrapper.UpdateFunctionConfigurationAsync(
            functionName,
            calculatorHandler,
            new Dictionary<string, string> { { "LOG_LEVEL", "DEBUG" } });

        do
        {
            config = await lambdaWrapper.GetFunctionAsync(functionName);
            Console.Write(".");
        }
        while (config.LastUpdateStatus == LastUpdateStatus.InProgress);

        uiWrapper.DisplayTitle("Call updated function");
        Console.WriteLine("Now call the updated function...");

        bool done = false;

        do
        {
            string? opSelected;

            Console.WriteLine("Select the operation to perform:");
            Console.WriteLine("\t1. add");
            Console.WriteLine("\t2. subtract");
            Console.WriteLine("\t3. multiply");
            Console.WriteLine("\t4. divide");
            Console.WriteLine("\tOr enter \"q\" to quit.");
            Console.WriteLine("Enter the number (1, 2, 3, 4, or q) of the
operation you want to perform: ");
            do
```

```
            {
                Console.Write("Your choice? ");
                opSelected = Console.ReadLine();
            }
            while (opSelected == string.Empty);

            var operation = (opSelected) switch
            {
                "1" => "add",
                "2" => "subtract",
                "3" => "multiply",
                "4" => "divide",
                "q" => "quit",
                _ => "add",
            };

            if (operation == "quit")
            {
                done = true;
            }
            else
            {
                // Get two numbers and an action from the user.
                value = string.Empty;
                do
                {
                    Console.Write("Enter the first value: ");
                    value = Console.ReadLine();
                }
                while (value == string.Empty);

                string? value2;
                do
                {
                    Console.Write("Enter a second value: ");
                    value2 = Console.ReadLine();
                }
                while (value2 == string.Empty);

                functionParameters = "{" +
                    "\"action\": \"" + operation + "\", " +
                    "\"x\": \"" + value + "\"," +
                    "\"y\": \"" + value2 + "\"" +
                "}";
```

```
                answer = await lambdaWrapper.InvokeFunctionAsync(functionName,
functionParameters);
                Console.WriteLine($"The answer when we {operation} the two
numbers is: {answer}.");
            }

            uiWrapper.PressEnter();
        } while (!done);

        // Delete the function created earlier.

        uiWrapper.DisplayTitle("Clean up resources");
        // Detach the IAM policy from the IAM role.
        Console.WriteLine("First detach the IAM policy from the role.");
        success = await lambdaRoleWrapper.DetachLambdaRolePolicyAsync(policyArn,
roleName);
        uiWrapper.WaitABit(15, "Let's wait for the policy to be fully detached
from the role.");

        Console.WriteLine("Delete the AWS Lambda function.");
        success = await lambdaWrapper.DeleteFunctionAsync(functionName);
        if (success)
        {
            Console.WriteLine($"The {functionName} function was deleted.");
        }
        else
        {
            Console.WriteLine($"Could not remove the function {functionName}");
        }

        // Now delete the IAM role created for use with the functions
        // created by the application.
        Console.WriteLine("Now we can delete the role that we created.");
        success = await lambdaRoleWrapper.DeleteLambdaRoleAsync(roleName);
        if (success)
        {
            Console.WriteLine("The role has been successfully removed.");
        }
        else
        {
            Console.WriteLine("Couldn't delete the role.");
        }
```

```
        Console.WriteLine("The Lambda Scenario is now complete.");
        uiWrapper.PressEnter();

        // Displays a formatted list of existing functions returned by the
        // LambdaMethods.ListFunctions.
        void DisplayFunctionList(List<FunctionConfiguration> functions)
        {
            functions.ForEach(functionConfig =>
            {

 Console.WriteLine($"{functionConfig.FunctionName}\t{functionConfig.Description}");
            });
        }
    }
}


namespace LambdaActions;

using Amazon.IdentityManagement;
using Amazon.IdentityManagement.Model;

public class LambdaRoleWrapper
{
    private readonly IAmazonIdentityManagementService _lambdaRoleService;

    public LambdaRoleWrapper(IAmazonIdentityManagementService lambdaRoleService)
    {
        _lambdaRoleService = lambdaRoleService;
    }

    /// <summary>
    /// Attach an AWS Identity and Access Management (IAM) role policy to the
    /// IAM role to be assumed by the AWS Lambda functions created for the
 scenario.
    /// </summary>
    /// <param name="policyArn">The Amazon Resource Name (ARN) of the IAM
 policy.</param>
    /// <param name="roleName">The name of the IAM role to attach the IAM policy
 to.</param>
    /// <returns>A Boolean value indicating the success of the action.</returns>
    public async Task<bool> AttachLambdaRolePolicyAsync(string policyArn, string
 roleName)
    {
```

```
        var response = await _lambdaRoleService.AttachRolePolicyAsync(new
 AttachRolePolicyRequest { PolicyArn = policyArn, RoleName = roleName });
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }

    /// <summary>
    /// Create a new IAM role.
    /// </summary>
    /// <param name="roleName">The name of the IAM role to create.</param>
    /// <param name="policyDocument">The policy document for the new IAM role.</
param>
    /// <returns>A string representing the ARN for newly created role.</returns>
    public async Task<string> CreateLambdaRoleAsync(string roleName, string
 policyDocument)
    {
        var request = new CreateRoleRequest
        {
            AssumeRolePolicyDocument = policyDocument,
            RoleName = roleName,
        };

        var response = await _lambdaRoleService.CreateRoleAsync(request);
        return response.Role.Arn;
    }

    /// <summary>
    /// Deletes an IAM role.
    /// </summary>
    /// <param name="roleName">The name of the role to delete.</param>
    /// <returns>A Boolean value indicating the success of the operation.</
returns>
    public async Task<bool> DeleteLambdaRoleAsync(string roleName)
    {
        var request = new DeleteRoleRequest
        {
            RoleName = roleName,
        };

        var response = await _lambdaRoleService.DeleteRoleAsync(request);
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }

    public async Task<bool> DetachLambdaRolePolicyAsync(string policyArn, string
 roleName)
```

```
    {
        var response = await _lambdaRoleService.DetachRolePolicyAsync(new
 DetachRolePolicyRequest { PolicyArn = policyArn, RoleName = roleName });
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }
}


namespace LambdaScenarioCommon;
public class UIWrapper
{
    public readonly string SepBar = new('-', Console.WindowWidth);

    /// <summary>
    /// Show information about the AWS Lambda Basics scenario.
    /// </summary>
    public void DisplayLambdaBasicsOverview()
    {
        Console.Clear();

        DisplayTitle("Welcome to AWS Lambda Basics");
        Console.WriteLine("This example application does the following:");
        Console.WriteLine("\t1. Creates an AWS Identity and Access Management
 (IAM) role that will be assumed by the functions we create.");
        Console.WriteLine("\t2. Attaches an IAM role policy that has Lambda
 permissions.");
        Console.WriteLine("\t3. Creates a Lambda function that increments the
 value passed to it.");
        Console.WriteLine("\t4. Calls the increment function and passes a
 value.");
        Console.WriteLine("\t5. Updates the code so that the function is a simple
 calculator.");
        Console.WriteLine("\t6. Calls the calculator function with the values
 entered.");
        Console.WriteLine("\t7. Deletes the Lambda function.");
        Console.WriteLine("\t7. Detaches the IAM role policy.");
        Console.WriteLine("\t8. Deletes the IAM role.");
        PressEnter();
    }

    /// <summary>
    /// Display a message and wait until the user presses enter.
    /// </summary>
    public void PressEnter()
```

```
    {
        Console.Write("\nPress <Enter> to continue. ");
        _ = Console.ReadLine();
        Console.WriteLine();
    }

    /// <summary>
    /// Pad a string with spaces to center it on the console display.
    /// </summary>
    /// <param name="strToCenter">The string to be centered.</param>
    /// <returns>The padded string.</returns>
    public string CenterString(string strToCenter)
    {
        var padAmount = (Console.WindowWidth - strToCenter.Length) / 2;
        var leftPad = new string(' ', padAmount);
        return $"{leftPad}{strToCenter}";
    }

    /// <summary>
    /// Display a line of hyphens, the centered text of the title and another
    /// line of hyphens.
    /// </summary>
    /// <param name="strTitle">The string to be displayed.</param>
    public void DisplayTitle(string strTitle)
    {
        Console.WriteLine(SepBar);
        Console.WriteLine(CenterString(strTitle));
        Console.WriteLine(SepBar);
    }

    /// <summary>
    /// Display a countdown and wait for a number of seconds.
    /// </summary>
    /// <param name="numSeconds">The number of seconds to wait.</param>
    public void WaitABit(int numSeconds, string msg)
    {
        Console.WriteLine(msg);

        // Wait for the requested number of seconds.
        for (int i = numSeconds; i > 0; i--)
        {
            System.Threading.Thread.Sleep(1000);
            Console.Write($"{i}...");
        }
```

```
        PressEnter();
    }
}
```

Définir un gestionnaire Lambda qui incrémente un nombre.

```
using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace LambdaIncrement;

public class Function
{

    /// <summary>
    /// A simple function increments the integer parameter.
    /// </summary>
    /// <param name="input">A JSON string containing an action, which must be
    /// "increment" and a string representing the value to increment.</param>
    /// <param name="context">The context object passed by Lambda containing
    /// information about invocation, function, and execution environment.</
param>
    /// <returns>A string representing the incremented value of the parameter.</
returns>
    public int FunctionHandler(Dictionary<string, string> input, ILambdaContext
 context)
    {
        if (input["action"] == "increment")
        {
            int inputValue = Convert.ToInt32(input["x"]);
            return inputValue + 1;
        }
        else
        {
            return 0;
        }
```

```
    }
}
```

Définir un deuxième gestionnaire Lambda qui effectue des opérations arithmétiques.

```
using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace LambdaCalculator;

public class Function
{

    /// <summary>
    /// A simple function that takes two number in string format and performs
    /// the requested arithmetic function.
    /// </summary>
    /// <param name="input">JSON data containing an action, and x and y values.
    /// Valid actions include: add, subtract, multiply, and divide.</param>
    /// <param name="context">The context object passed by Lambda containing
    /// information about invocation, function, and execution environment.</
param>
    /// <returns>A string representing the results of the calculation.</returns>
    public int FunctionHandler(Dictionary<string, string> input, ILambdaContext
 context)
    {
        var action = input["action"];
        int x = Convert.ToInt32(input["x"]);
        int y = Convert.ToInt32(input["y"]);
        int result;
        switch (action)
        {
            case "add":
                result = x + y;
                break;
            case "subtract":
                result = x - y;
```

```
                break;
            case "multiply":
                result = x * y;
                break;
            case "divide":
                if (y == 0)
                {
                    Console.Error.WriteLine("Divide by zero error.");
                    result = 0;
                }
                else
                    result = x / y;
                break;
            default:
                Console.Error.WriteLine($"{action} is not a valid operation.");
                result = 0;
                break;
        }
        return result;
    }
}
```

- Pour plus d'informations sur l'API, consultez les rubriques suivantes dans la référence de l'API AWS SDK pour .NET .
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

C++

SDK pour C++

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
//! Get started with functions scenario.
/*!
 \param clientConfig: AWS client configuration.
 \return bool: Successful completion.
 */
bool AwsDoc::Lambda::getStartedWithFunctionsScenario(
        const Aws::Client::ClientConfiguration &clientConfig) {

    Aws::Lambda::LambdaClient client(clientConfig);

    // 1. Create an AWS Identity and Access Management (IAM) role for Lambda
 function.
    Aws::String roleArn;
    if (!getIamRoleArn(roleArn, clientConfig)) {
        return false;
    }

    // 2. Create a Lambda function.
    int seconds = 0;
    do {
        Aws::Lambda::Model::CreateFunctionRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetDescription(LAMBDA_DESCRIPTION); // Optional.
#if USE_CPP_LAMBDA_FUNCTION
        request.SetRuntime(Aws::Lambda::Model::Runtime::provided_al2);
        request.SetTimeout(15);
        request.SetMemorySize(128);

        // Assume the AWS Lambda function was built in Docker with same
 architecture
        // as this code.
#if  defined(__x86_64__)
```

```
        request.SetArchitectures({Aws::Lambda::Model::Architecture::x86_64});
#elif defined(__aarch64__)
        request.SetArchitectures({Aws::Lambda::Model::Architecture::arm64});
#else
#error "Unimplemented architecture"
#endif // defined(architecture)
#else
        request.SetRuntime(Aws::Lambda::Model::Runtime::python3_9);
#endif
        request.SetRole(roleArn);
        request.SetHandler(LAMBDA_HANDLER_NAME);
        request.SetPublish(true);
        Aws::Lambda::Model::FunctionCode code;
        std::ifstream ifstream(INCREMENT_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteIamRole(clientConfig);
            return false;
        }

        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();

        code.SetZipFile(Aws::Utils::ByteBuffer((unsigned char *)
 buffer.str().c_str(),
                                               buffer.str().length()));
        request.SetCode(code);

        Aws::Lambda::Model::CreateFunctionOutcome outcome =
 client.CreateFunction(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda function was successfully created. " <<
 seconds
```

```
                    << " seconds elapsed." << std::endl;
            break;
        }
        else if (outcome.GetError().GetErrorType() ==
                 Aws::Lambda::LambdaErrors::INVALID_PARAMETER_VALUE &&
                 outcome.GetError().GetMessage().find("role") >= 0) {
            if ((seconds % 5) == 0) { // Log status every 10 seconds.
                std::cout
                        << "Waiting for the IAM role to become available as a
CreateFunction parameter. "
                        << seconds
                        << " seconds elapsed." << std::endl;

                std::cout << outcome.GetError().GetMessage() << std::endl;
            }
        }
        else {
            std::cerr << "Error with CreateFunction. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            deleteIamRole(clientConfig);
            return false;
        }
        ++seconds;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    } while (60 > seconds);

    std::cout << "The current Lambda function increments 1 by an input." <<
std::endl;

    // 3.  Invoke the Lambda function.
    {
        int increment = askQuestionForInt("Enter an increment integer: ");

        Aws::Lambda::Model::InvokeResult invokeResult;
        Aws::Utils::Json::JsonValue jsonPayload;
        jsonPayload.WithString("action", "increment");
        jsonPayload.WithInteger("number", increment);
        if (invokeLambdaFunction(jsonPayload, Aws::Lambda::Model::LogType::Tail,
                                 invokeResult, client)) {
            Aws::Utils::Json::JsonValue jsonValue(invokeResult.GetPayload());
            Aws::Map<Aws::String, Aws::Utils::Json::JsonView> values =
                    jsonValue.View().GetAllObjects();
            auto iter = values.find("result");
```

```
            if (iter != values.end() && iter->second.IsIntegerType()) {
                {
                    std::cout << INCREMENT_RESUlT_PREFIX
                              << iter->second.AsInteger() << std::endl;
                }
            }
            else {
                std::cout << "There was an error in execution. Here is the log."
                          << std::endl;
                Aws::Utils::ByteBuffer buffer =
 Aws::Utils::HashingUtils::Base64Decode(
                        invokeResult.GetLogResult());
                std::cout << "With log " << buffer.GetUnderlyingData() <<
 std::endl;
            }
        }
    }

    std::cout
            << "The Lambda function will now be updated with new code. Press
 return to continue, ";
    Aws::String answer;
    std::getline(std::cin, answer);

    // 4.  Update the Lambda function code.
    {
        Aws::Lambda::Model::UpdateFunctionCodeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        std::ifstream ifstream(CALCULATOR_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteLambdaFunction(client);
            deleteIamRole(clientConfig);
            return false;
        }
```

```
        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();
        request.SetZipFile(
                Aws::Utils::ByteBuffer((unsigned char *) buffer.str().c_str(),
                                       buffer.str().length()));
        request.SetPublish(true);

        Aws::Lambda::Model::UpdateFunctionCodeOutcome outcome =
client.UpdateFunctionCode(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda code was successfully updated." <<
std::endl;
        }
        else {
            std::cerr << "Error with Lambda::UpdateFunctionCode. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
    }

    std::cout
              << "This function uses an environment variable to control the logging
level."
              << std::endl;
    std::cout
              << "UpdateFunctionConfiguration will be used to set the LOG_LEVEL to
DEBUG."
              << std::endl;
    seconds = 0;

    // 5.  Update the Lambda function configuration.
    do {
        ++seconds;
        std::this_thread::sleep_for(std::chrono::seconds(1));
        Aws::Lambda::Model::UpdateFunctionConfigurationRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        Aws::Lambda::Model::Environment environment;
        environment.AddVariables("LOG_LEVEL", "DEBUG");
        request.SetEnvironment(environment);
```

```
        Aws::Lambda::Model::UpdateFunctionConfigurationOutcome outcome =
 client.UpdateFunctionConfiguration(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda configuration was successfully updated."
                      << std::endl;
            break;
        }

            // RESOURCE_IN_USE: function code update not completed.
        else if (outcome.GetError().GetErrorType() !=
                 Aws::Lambda::LambdaErrors::RESOURCE_IN_USE) {
            if ((seconds % 10) == 0) { // Log status every 10 seconds.
                std::cout << "Lambda function update in progress . After " <<
 seconds
                          << " seconds elapsed." << std::endl;
            }
        }
        else {
            std::cerr << "Error with Lambda::UpdateFunctionConfiguration. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }

    } while (0 < seconds);

    if (0 > seconds) {
        std::cerr << "Function failed to become active." << std::endl;
    }
    else {
        std::cout << "Updated function active after " << seconds << " seconds."
                  << std::endl;
    }

    std::cout
            << "\nThe new code applies an arithmetic operator to two variables, x
 an y."
            << std::endl;
    std::vector<Aws::String> operators = {"plus", "minus", "times", "divided-
by"};
    for (size_t i = 0; i < operators.size(); ++i) {
        std::cout << "   " << i + 1 << " " << operators[i] << std::endl;
    }
```

```
   // 6.  Invoke the updated Lambda function.
   do {
       int operatorIndex = askQuestionForIntRange("Select an operator index 1 -
4 ", 1,
                                                  4);
       int x = askQuestionForInt("Enter an integer for the x value ");
       int y = askQuestionForInt("Enter an integer for the y value ");

       Aws::Utils::Json::JsonValue calculateJsonPayload;
       calculateJsonPayload.WithString("action", operators[operatorIndex - 1]);
       calculateJsonPayload.WithInteger("x", x);
       calculateJsonPayload.WithInteger("y", y);
       Aws::Lambda::Model::InvokeResult calculatedResult;
       if (invokeLambdaFunction(calculateJsonPayload,
                                Aws::Lambda::Model::LogType::Tail,
                                calculatedResult, client)) {
           Aws::Utils::Json::JsonValue jsonValue(calculatedResult.GetPayload());
           Aws::Map<Aws::String, Aws::Utils::Json::JsonView> values =
                   jsonValue.View().GetAllObjects();
           auto iter = values.find("result");
           if (iter != values.end() && iter->second.IsIntegerType()) {
               std::cout << ARITHMETIC_RESULT_PREFIX << x << " "
                         << operators[operatorIndex - 1] << " "
                         << y << " is " << iter->second.AsInteger() <<
std::endl;
           }
           else if (iter != values.end() && iter->second.IsFloatingPointType())
{
               std::cout << ARITHMETIC_RESULT_PREFIX << x << " "
                         << operators[operatorIndex - 1] << " "
                         << y << " is " << iter->second.AsDouble() << std::endl;
           }
           else {
               std::cout << "There was an error in execution. Here is the log."
                         << std::endl;
               Aws::Utils::ByteBuffer buffer =
Aws::Utils::HashingUtils::Base64Decode(
                        calculatedResult.GetLogResult());
               std::cout << "With log " << buffer.GetUnderlyingData() <<
std::endl;
           }
       }
```

```
        answer = askQuestion("Would you like to try another operation? (y/n) ");
    } while (answer == "y");

    std::cout
            << "A list of the lambda functions will be retrieved. Press return to
continue, ";
    std::getline(std::cin, answer);

    // 7.  List the Lambda functions.

    std::vector<Aws::String> functions;
    Aws::String marker;

    do {
        Aws::Lambda::Model::ListFunctionsRequest request;
        if (!marker.empty()) {
            request.SetMarker(marker);
        }

        Aws::Lambda::Model::ListFunctionsOutcome outcome = client.ListFunctions(
                request);

        if (outcome.IsSuccess()) {
            const Aws::Lambda::Model::ListFunctionsResult &result =
outcome.GetResult();
            std::cout << result.GetFunctions().size()
                      << " lambda functions were retrieved." << std::endl;

            for (const Aws::Lambda::Model::FunctionConfiguration
&functionConfiguration: result.GetFunctions()) {
                functions.push_back(functionConfiguration.GetFunctionName());
                std::cout << functions.size() << "  "
                          << functionConfiguration.GetDescription() << std::endl;
                std::cout << "   "
                          <<
Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(
                                  functionConfiguration.GetRuntime()) << ": "
                          << functionConfiguration.GetHandler()
                          << std::endl;
            }
            marker = result.GetNextMarker();
        }
        else {
            std::cerr << "Error with Lambda::ListFunctions. "
```

```
                    << outcome.GetError().GetMessage()
                    << std::endl;
        }
    } while (!marker.empty());

    // 8.  Get a Lambda function.
    if (!functions.empty()) {
        std::stringstream question;
        question << "Choose a function to retrieve between 1 and " <<
functions.size()
                 << " ";
        int functionIndex = askQuestionForIntRange(question.str(), 1,

static_cast<int>(functions.size()));

        Aws::String functionName = functions[functionIndex - 1];

        Aws::Lambda::Model::GetFunctionRequest request;
        request.SetFunctionName(functionName);

        Aws::Lambda::Model::GetFunctionOutcome outcome =
client.GetFunction(request);

        if (outcome.IsSuccess()) {
            std::cout << "Function retrieve.\n" <<

outcome.GetResult().GetConfiguration().Jsonize().View().WriteReadable()
                    << std::endl;
        }
        else {
            std::cerr << "Error with Lambda::GetFunction. "
                    << outcome.GetError().GetMessage()
                    << std::endl;
        }
    }

    std::cout << "The resources will be deleted. Press return to continue, ";
    std::getline(std::cin, answer);

    // 9.  Delete the Lambda function.
    bool result = deleteLambdaFunction(client);

    // 10. Delete the IAM role.
    return result && deleteIamRole(clientConfig);
```

```
}

//! Routine which invokes a Lambda function and returns the result.
/*!
 \param jsonPayload: Payload for invoke function.
 \param logType: Log type setting for invoke function.
 \param invokeResult: InvokeResult object to receive the result.
 \param client: Lambda client.
 \return bool: Successful completion.
 */
bool
AwsDoc::Lambda::invokeLambdaFunction(const Aws::Utils::Json::JsonValue
 &jsonPayload,
                                     Aws::Lambda::Model::LogType logType,
                                     Aws::Lambda::Model::InvokeResult
 &invokeResult,
                                     const Aws::Lambda::LambdaClient &client) {
    int seconds = 0;
    bool result = false;
    /*
     * In this example, the Invoke function can be called before recently created
 resources are
     * available.  The Invoke function is called repeatedly until the resources
 are
     * available.
     */
    do {
        Aws::Lambda::Model::InvokeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetLogType(logType);
        std::shared_ptr<Aws::IOStream> payload =
 Aws::MakeShared<Aws::StringStream>(
                "FunctionTest");
        *payload << jsonPayload.View().WriteReadable();
        request.SetBody(payload);
        request.SetContentType("application/json");
        Aws::Lambda::Model::InvokeOutcome outcome = client.Invoke(request);

        if (outcome.IsSuccess()) {
            invokeResult = std::move(outcome.GetResult());
            result = true;
            break;
        }
```

```
            // ACCESS_DENIED: because the role is not available yet.
            // RESOURCE_CONFLICT: because the Lambda function is being created or
 updated.
        else if ((outcome.GetError().GetErrorType() ==
                  Aws::Lambda::LambdaErrors::ACCESS_DENIED) ||
                 (outcome.GetError().GetErrorType() ==
                  Aws::Lambda::LambdaErrors::RESOURCE_CONFLICT)) {
            if ((seconds % 5) == 0) { // Log status every 10 seconds.
                std::cout << "Waiting for the invoke api to be available, status
 " <<
                          ((outcome.GetError().GetErrorType() ==
                            Aws::Lambda::LambdaErrors::ACCESS_DENIED ?
                            "ACCESS_DENIED" : "RESOURCE_CONFLICT")) << ". " <<
 seconds
                          << " seconds elapsed." << std::endl;
            }
        }
        else {
            std::cerr << "Error with Lambda::InvokeRequest. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            break;
        }
        ++seconds;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    } while (seconds < 60);

    return result;
}
```

- Pour plus de détails sur l'API, consultez les rubriques suivantes dans la Référence des API du kit AWS SDK pour C++ .
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

Créez un scénario interactif qui vous montre comment démarrer avec les fonctions Lambda.

```
import (
 "archive/zip"
 "bytes"
 "context"
 "encoding/base64"
 "encoding/json"
 "errors"
 "fmt"
 "log"
 "os"
 "strings"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/iam"
 iamtypes "github.com/aws/aws-sdk-go-v2/service/iam/types"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/lambda/actions"
)

// GetStartedFunctionsScenario shows you how to use AWS Lambda to perform the
 following
// actions:
//
//  1. Create an AWS Identity and Access Management (IAM) role and Lambda
 function, then upload handler code.
//  2. Invoke the function with a single parameter and get results.
//  3. Update the function code and configure with an environment variable.
```

```
//  4. Invoke the function with new parameters and get results. Display the
 returned execution log.
//  5. List the functions for your account, then clean up resources.
type GetStartedFunctionsScenario struct {
 sdkConfig       aws.Config
 functionWrapper actions.FunctionWrapper
 questioner      demotools.IQuestioner
 helper          IScenarioHelper
 isTestRun       bool
}

// NewGetStartedFunctionsScenario constructs a GetStartedFunctionsScenario
 instance from a configuration.
// It uses the specified config to get a Lambda client and create wrappers for
 the actions
// used in the scenario.
func NewGetStartedFunctionsScenario(sdkConfig aws.Config, questioner
 demotools.IQuestioner,
 helper IScenarioHelper) GetStartedFunctionsScenario {
 lambdaClient := lambda.NewFromConfig(sdkConfig)
 return GetStartedFunctionsScenario{
  sdkConfig:       sdkConfig,
  functionWrapper: actions.FunctionWrapper{LambdaClient: lambdaClient},
  questioner:      questioner,
  helper:          helper,
 }
}

// Run runs the interactive scenario.
func (scenario GetStartedFunctionsScenario) Run(ctx context.Context) {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong with the demo.\n")
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Println("Welcome to the AWS Lambda get started with functions demo.")
 log.Println(strings.Repeat("-", 88))

 role := scenario.GetOrCreateRole(ctx)
 funcName := scenario.CreateFunction(ctx, role)
 scenario.InvokeIncrement(ctx, funcName)
 scenario.UpdateFunction(ctx, funcName)
```

```
 scenario.InvokeCalculator(ctx, funcName)
 scenario.ListFunctions(ctx)
 scenario.Cleanup(ctx, role, funcName)

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}

// GetOrCreateRole checks whether the specified role exists and returns it if it
 does.
// Otherwise, a role is created that specifies Lambda as a trusted principal.
// The AWSLambdaBasicExecutionRole managed policy is attached to the role and the
 role
// is returned.
func (scenario GetStartedFunctionsScenario) GetOrCreateRole(ctx context.Context)
 *iamtypes.Role {
 var role *iamtypes.Role
 iamClient := iam.NewFromConfig(scenario.sdkConfig)
 log.Println("First, we need an IAM role that Lambda can assume.")
 roleName := scenario.questioner.Ask("Enter a name for the role:",
 demotools.NotEmpty{})
 getOutput, err := iamClient.GetRole(ctx, &iam.GetRoleInput{
  RoleName: aws.String(roleName)})
 if err != nil {
  var noSuch *iamtypes.NoSuchEntityException
  if errors.As(err, &noSuch) {
   log.Printf("Role %v doesn't exist. Creating it....\n", roleName)
  } else {
   log.Panicf("Couldn't check whether role %v exists. Here's why: %v\n",
    roleName, err)
  }
 } else {
  role = getOutput.Role
  log.Printf("Found role %v.\n", *role.RoleName)
 }
 if role == nil {
  trustPolicy := PolicyDocument{
   Version: "2012-10-17",
   Statement: []PolicyStatement{{
    Effect:    "Allow",
    Principal: map[string]string{"Service": "lambda.amazonaws.com"},
    Action:    []string{"sts:AssumeRole"},
   }},
```

```
  }
  policyArn := "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"
  createOutput, err := iamClient.CreateRole(ctx, &iam.CreateRoleInput{
   AssumeRolePolicyDocument: aws.String(trustPolicy.String()),
   RoleName:                 aws.String(roleName),
  })
  if err != nil {
   log.Panicf("Couldn't create role %v. Here's why: %v\n", roleName, err)
  }
  role = createOutput.Role
  _, err = iamClient.AttachRolePolicy(ctx, &iam.AttachRolePolicyInput{
   PolicyArn: aws.String(policyArn),
   RoleName:  aws.String(roleName),
  })
  if err != nil {
   log.Panicf("Couldn't attach a policy to role %v. Here's why: %v\n", roleName,
 err)
  }
  log.Printf("Created role %v.\n", *role.RoleName)
  log.Println("Let's give AWS a few seconds to propagate resources...")
  scenario.helper.Pause(10)
 }
 log.Println(strings.Repeat("-", 88))
 return role
}

// CreateFunction creates a Lambda function and uploads a handler written in
 Python.
// The code for the Python handler is packaged as a []byte in .zip format.
func (scenario GetStartedFunctionsScenario) CreateFunction(ctx context.Context,
 role *iamtypes.Role) string {
 log.Println("Let's create a function that increments a number.\n" +
  "The function uses the 'lambda_handler_basic.py' script found in the \n" +
  "'handlers' directory of this project.")
 funcName := scenario.questioner.Ask("Enter a name for the Lambda function:",
 demotools.NotEmpty{})
 zipPackage := scenario.helper.CreateDeploymentPackage("lambda_handler_basic.py",
 fmt.Sprintf("%v.py", funcName))
 log.Printf("Creating function %v and waiting for it to be ready.", funcName)
 funcState := scenario.functionWrapper.CreateFunction(ctx, funcName,
 fmt.Sprintf("%v.lambda_handler", funcName),
  role.Arn, zipPackage)
 log.Printf("Your function is %v.", funcState)
 log.Println(strings.Repeat("-", 88))
```

```
 return funcName
}

// InvokeIncrement invokes a Lambda function that increments a number. The
 function
// parameters are contained in a Go struct that is used to serialize the
 parameters to
// a JSON payload that is passed to the function.
// The result payload is deserialized into a Go struct that contains an int
 value.
func (scenario GetStartedFunctionsScenario) InvokeIncrement(ctx context.Context,
 funcName string) {
 parameters := actions.IncrementParameters{Action: "increment"}
 log.Println("Let's invoke our function. This function increments a number.")
 parameters.Number = scenario.questioner.AskInt("Enter a number to increment:",
 demotools.NotEmpty{})
 log.Printf("Invoking %v with %v...\n", funcName, parameters.Number)
 invokeOutput := scenario.functionWrapper.Invoke(ctx, funcName, parameters,
 false)
 var payload actions.LambdaResultInt
 err := json.Unmarshal(invokeOutput.Payload, &payload)
 if err != nil {
  log.Panicf("Couldn't unmarshal payload from invoking %v. Here's why: %v\n",
   funcName, err)
 }
 log.Printf("Invoking %v with %v returned %v.\n", funcName, parameters.Number,
 payload)
 log.Println(strings.Repeat("-", 88))
}

// UpdateFunction updates the code for a Lambda function by uploading a simple
 arithmetic
// calculator written in Python. The code for the Python handler is packaged as a
// []byte in .zip format.
// After the code is updated, the configuration is also updated with a new log
// level that instructs the handler to log additional information.
func (scenario GetStartedFunctionsScenario) UpdateFunction(ctx context.Context,
 funcName string) {
 log.Println("Let's update the function to an arithmetic calculator.\n" +
  "The function uses the 'lambda_handler_calculator.py' script found in the \n" +
  "'handlers' directory of this project.")
 scenario.questioner.Ask("Press Enter when you're ready.")
 log.Println("Creating deployment package...")
```

```
 zipPackage :=
 scenario.helper.CreateDeploymentPackage("lambda_handler_calculator.py",
  fmt.Sprintf("%v.py", funcName))
 log.Println("...and updating the Lambda function and waiting for it to be
 ready.")
 funcState := scenario.functionWrapper.UpdateFunctionCode(ctx, funcName,
 zipPackage)
 log.Printf("Updated function %v. Its current state is %v.", funcName, funcState)
 log.Println("This function uses an environment variable to control logging
 level.")
 log.Println("Let's set it to DEBUG to get the most logging.")
 scenario.functionWrapper.UpdateFunctionConfiguration(ctx, funcName,
  map[string]string{"LOG_LEVEL": "DEBUG"})
 log.Println(strings.Repeat("-", 88))
}

// InvokeCalculator invokes the Lambda calculator function. The parameters are
 stored in a
// Go struct that is used to serialize the parameters to a JSON payload. That
 payload is then passed
// to the function.
// The result payload is deserialized to a Go struct that stores the result as
 either an
// int or float32, depending on the kind of operation that was specified.
func (scenario GetStartedFunctionsScenario) InvokeCalculator(ctx context.Context,
 funcName string) {
 wantInvoke := true
 choices := []string{"plus", "minus", "times", "divided-by"}
 for wantInvoke {
  choice := scenario.questioner.AskChoice("Select an arithmetic operation:\n",
 choices)
  x := scenario.questioner.AskInt("Enter a value for x:", demotools.NotEmpty{})
  y := scenario.questioner.AskInt("Enter a value for y:", demotools.NotEmpty{})
  log.Printf("Invoking %v %v %v...", x, choices[choice], y)
  calcParameters := actions.CalculatorParameters{
   Action: choices[choice],
   X:      x,
   Y:      y,
  }
  invokeOutput := scenario.functionWrapper.Invoke(ctx, funcName, calcParameters,
 true)
  var payload any
  if choice == 3 { // divide-by results in a float.
   payload = actions.LambdaResultFloat{}
```

```
  } else {
   payload = actions.LambdaResultInt{}
  }
  err := json.Unmarshal(invokeOutput.Payload, &payload)
  if err != nil {
   log.Panicf("Couldn't unmarshal payload from invoking %v. Here's why: %v\n",
    funcName, err)
  }
  log.Printf("Invoking %v with %v %v %v returned %v.\n", funcName,
   calcParameters.X, calcParameters.Action, calcParameters.Y, payload)
  scenario.questioner.Ask("Press Enter to see the logs from the call.")
  logRes, err := base64.StdEncoding.DecodeString(*invokeOutput.LogResult)
  if err != nil {
   log.Panicf("Couldn't decode log result. Here's why: %v\n", err)
  }
  log.Println(string(logRes))
  wantInvoke = scenario.questioner.AskBool("Do you want to calculate again? (y/
n)", "y")
 }
 log.Println(strings.Repeat("-", 88))
}

// ListFunctions lists up to the specified number of functions for your account.
func (scenario GetStartedFunctionsScenario) ListFunctions(ctx context.Context) {
 count := scenario.questioner.AskInt(
  "Let's list functions for your account. How many do you want to see?",
 demotools.NotEmpty{})
 functions := scenario.functionWrapper.ListFunctions(ctx, count)
 log.Printf("Found %v functions:", len(functions))
 for _, function := range functions {
  log.Printf("\t%v", *function.FunctionName)
 }
 log.Println(strings.Repeat("-", 88))
}

// Cleanup removes the IAM and Lambda resources created by the example.
func (scenario GetStartedFunctionsScenario) Cleanup(ctx context.Context, role
 *iamtypes.Role, funcName string) {
 if scenario.questioner.AskBool("Do you want to clean up resources created for
 this example? (y/n)",
  "y") {
  iamClient := iam.NewFromConfig(scenario.sdkConfig)
  policiesOutput, err := iamClient.ListAttachedRolePolicies(ctx,
   &iam.ListAttachedRolePoliciesInput{RoleName: role.RoleName})
```

```
  if err != nil {
   log.Panicf("Couldn't get policies attached to role %v. Here's why: %v\n",
    *role.RoleName, err)
  }
  for _, policy := range policiesOutput.AttachedPolicies {
   _, err = iamClient.DetachRolePolicy(ctx, &iam.DetachRolePolicyInput{
    PolicyArn: policy.PolicyArn, RoleName: role.RoleName,
   })
   if err != nil {
    log.Panicf("Couldn't detach policy %v from role %v. Here's why: %v\n",
     *policy.PolicyArn, *role.RoleName, err)
   }
  }
  _, err = iamClient.DeleteRole(ctx, &iam.DeleteRoleInput{RoleName:
 role.RoleName})
  if err != nil {
   log.Panicf("Couldn't delete role %v. Here's why: %v\n", *role.RoleName, err)
  }
  log.Printf("Deleted role %v.\n", *role.RoleName)

  scenario.functionWrapper.DeleteFunction(ctx, funcName)
  log.Printf("Deleted function %v.\n", funcName)
 } else {
  log.Println("Okay. Don't forget to delete the resources when you're done with
 them.")
 }
}

// IScenarioHelper abstracts I/O and wait functions from a scenario so that they
// can be mocked for unit testing.
type IScenarioHelper interface {
 Pause(secs int)
 CreateDeploymentPackage(sourceFile string, destinationFile string) *bytes.Buffer
}

// ScenarioHelper lets the caller specify the path to Lambda handler functions.
type ScenarioHelper struct {
 HandlerPath string
}

// Pause waits for the specified number of seconds.
func (helper *ScenarioHelper) Pause(secs int) {
 time.Sleep(time.Duration(secs) * time.Second)
}
```

```
// CreateDeploymentPackage creates an AWS Lambda deployment package from a source
 file. The
// deployment package is stored in .zip format in a bytes.Buffer. The buffer can
 be
// used to pass a []byte to Lambda when creating the function.
// The specified destinationFile is the name to give the file when it's deployed
 to Lambda.
func (helper *ScenarioHelper) CreateDeploymentPackage(sourceFile string,
 destinationFile string) *bytes.Buffer {
 var err error
 buffer := &bytes.Buffer{}
 writer := zip.NewWriter(buffer)
 zFile, err := writer.Create(destinationFile)
 if err != nil {
  log.Panicf("Couldn't create destination archive %v. Here's why: %v\n",
 destinationFile, err)
 }
 sourceBody, err := os.ReadFile(fmt.Sprintf("%v/%v", helper.HandlerPath,
 sourceFile))
 if err != nil {
  log.Panicf("Couldn't read handler source file %v. Here's why: %v\n",
   sourceFile, err)
 } else {
  _, err = zFile.Write(sourceBody)
  if err != nil {
   log.Panicf("Couldn't write handler %v to zip archive. Here's why: %v\n",
    sourceFile, err)
  }
 }
 err = writer.Close()
 if err != nil {
  log.Panicf("Couldn't close zip writer. Here's why: %v\n", err)
 }
 return buffer
}
```

Créez une structure qui englobe les actions Lambda individuelles.

```
import (
```

```
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}


// GetFunction gets data about the Lambda function specified by functionName.
func (wrapper FunctionWrapper) GetFunction(ctx context.Context, functionName
 string) types.State {
 var state types.State
 funcOutput, err := wrapper.LambdaClient.GetFunction(ctx,
 &lambda.GetFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't get function %v. Here's why: %v\n", functionName, err)
 } else {
  state = funcOutput.Configuration.State
 }
 return state
}



// CreateFunction creates a new Lambda function from code contained in the
 zipPackage
// buffer. The specified handlerName must match the name of the file and function
// contained in the uploaded code. The role specified by iamRoleArn is assumed by
// Lambda and grants specific permissions.
// When the function already exists, types.StateActive is returned.
```

```
// When the function is created, a lambda.FunctionActiveV2Waiter is used to wait
 until the
// function is active.
func (wrapper FunctionWrapper) CreateFunction(ctx context.Context, functionName
 string, handlerName string,
 iamRoleArn *string, zipPackage *bytes.Buffer) types.State {
 var state types.State
 _, err := wrapper.LambdaClient.CreateFunction(ctx, &lambda.CreateFunctionInput{
  Code:         &types.FunctionCode{ZipFile: zipPackage.Bytes()},
  FunctionName: aws.String(functionName),
  Role:         iamRoleArn,
  Handler:      aws.String(handlerName),
  Publish:      true,
  Runtime:      types.RuntimePython39,
 })
 if err != nil {
  var resConflict *types.ResourceConflictException
  if errors.As(err, &resConflict) {
   log.Printf("Function %v already exists.\n", functionName)
   state = types.StateActive
  } else {
   log.Panicf("Couldn't create function %v. Here's why: %v\n", functionName, err)
  }
 } else {
  waiter := lambda.NewFunctionActiveV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(ctx, &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
 functionName, err)
  } else {
   state = funcOutput.Configuration.State
  }
 }
 return state
}



// UpdateFunctionCode updates the code for the Lambda function specified by
 functionName.
// The existing code for the Lambda function is entirely replaced by the code in
 the
```

```
// zipPackage buffer. After the update action is called, a
 lambda.FunctionUpdatedV2Waiter
// is used to wait until the update is successful.
func (wrapper FunctionWrapper) UpdateFunctionCode(ctx context.Context,
 functionName string, zipPackage *bytes.Buffer) types.State {
 var state types.State
 _, err := wrapper.LambdaClient.UpdateFunctionCode(ctx,
 &lambda.UpdateFunctionCodeInput{
  FunctionName: aws.String(functionName), ZipFile: zipPackage.Bytes(),
 })
 if err != nil {
  log.Panicf("Couldn't update code for function %v. Here's why: %v\n",
 functionName, err)
 } else {
  waiter := lambda.NewFunctionUpdatedV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(ctx, &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
 functionName, err)
  } else {
   state = funcOutput.Configuration.State
  }
 }
 return state
}



// UpdateFunctionConfiguration updates a map of environment variables configured
 for
// the Lambda function specified by functionName.
func (wrapper FunctionWrapper) UpdateFunctionConfiguration(ctx context.Context,
 functionName string, envVars map[string]string) {
 _, err := wrapper.LambdaClient.UpdateFunctionConfiguration(ctx,
 &lambda.UpdateFunctionConfigurationInput{
  FunctionName: aws.String(functionName),
  Environment:  &types.Environment{Variables: envVars},
 })
 if err != nil {
  log.Panicf("Couldn't update configuration for %v. Here's why: %v",
 functionName, err)
 }
}
```

```
// ListFunctions lists up to maxItems functions for the account. This function
 uses a
// lambda.ListFunctionsPaginator to paginate the results.
func (wrapper FunctionWrapper) ListFunctions(ctx context.Context, maxItems int)
 []types.FunctionConfiguration {
 var functions []types.FunctionConfiguration
 paginator := lambda.NewListFunctionsPaginator(wrapper.LambdaClient,
 &lambda.ListFunctionsInput{
  MaxItems: aws.Int32(int32(maxItems)),
 })
 for paginator.HasMorePages() && len(functions) < maxItems {
  pageOutput, err := paginator.NextPage(ctx)
  if err != nil {
   log.Panicf("Couldn't list functions for your account. Here's why: %v\n", err)
  }
  functions = append(functions, pageOutput.Functions...)
 }
 return functions
}



// DeleteFunction deletes the Lambda function specified by functionName.
func (wrapper FunctionWrapper) DeleteFunction(ctx context.Context, functionName
 string) {
 _, err := wrapper.LambdaClient.DeleteFunction(ctx, &lambda.DeleteFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't delete function %v. Here's why: %v\n", functionName, err)
 }
}



// Invoke invokes the Lambda function specified by functionName, passing the
 parameters
// as a JSON payload. When getLog is true, types.LogTypeTail is specified, which
 tells
// Lambda to include the last few log lines in the returned result.
```

```
func (wrapper FunctionWrapper) Invoke(ctx context.Context, functionName string,
 parameters any, getLog bool) *lambda.InvokeOutput {
 logType := types.LogTypeNone
 if getLog {
  logType = types.LogTypeTail
 }
 payload, err := json.Marshal(parameters)
 if err != nil {
  log.Panicf("Couldn't marshal parameters to JSON. Here's why %v\n", err)
 }
 invokeOutput, err := wrapper.LambdaClient.Invoke(ctx, &lambda.InvokeInput{
  FunctionName: aws.String(functionName),
  LogType:      logType,
  Payload:      payload,
 })
 if err != nil {
  log.Panicf("Couldn't invoke function %v. Here's why: %v\n", functionName, err)
 }
 return invokeOutput
}



// IncrementParameters is used to serialize parameters to the increment Lambda
 handler.
type IncrementParameters struct {
 Action string `json:"action"`
 Number int    `json:"number"`
}

// CalculatorParameters is used to serialize parameters to the calculator Lambda
 handler.
type CalculatorParameters struct {
 Action string `json:"action"`
 X      int    `json:"x"`
 Y      int    `json:"y"`
}

// LambdaResultInt is used to deserialize an int result from a Lambda handler.
type LambdaResultInt struct {
 Result int `json:"result"`
}
```

```
// LambdaResultFloat is used to deserialize a float32 result from a Lambda
 handler.
type LambdaResultFloat struct {
 Result float32 `json:"result"`
}
```

Définir un gestionnaire Lambda qui incrémente un nombre.

```
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)


def lambda_handler(event, context):
    """
    Accepts an action and a single number, performs the specified action on the
 number,
    and returns the result. The only allowable action is 'increment'.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the action.
    """
    result = None
    action = event.get("action")
    if action == "increment":
        result = event.get("number", 0) + 1
        logger.info("Calculated result of %s", result)
    else:
        logger.error("%s is not a valid action.", action)

    response = {"result": result}
    return response
```

Définir un deuxième gestionnaire Lambda qui effectue des opérations arithmétiques.

```
import logging
import os


logger = logging.getLogger()

# Define a list of Python lambda functions that are called by this AWS Lambda
 function.
ACTIONS = {
    "plus": lambda x, y: x + y,
    "minus": lambda x, y: x - y,
    "times": lambda x, y: x * y,
    "divided-by": lambda x, y: x / y,
}


def lambda_handler(event, context):
    """
    Accepts an action and two numbers, performs the specified action on the
 numbers,
    and returns the result.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the specified action.
    """
    # Set the log level based on a variable configured in the Lambda environment.
    logger.setLevel(os.environ.get("LOG_LEVEL", logging.INFO))
    logger.debug("Event: %s", event)

    action = event.get("action")
    func = ACTIONS.get(action)
    x = event.get("x")
    y = event.get("y")
    result = None
    try:
        if func is not None and x is not None and y is not None:
            result = func(x, y)
            logger.info("%s %s %s is %s", x, action, y, result)
        else:
            logger.error("I can't calculate %s %s %s.", x, action, y)
```

```
    except ZeroDivisionError:
        logger.warning("I can't divide %s by 0!", x)

    response = {"result": result}
    return response
```

- Pour plus d'informations sur l'API, consultez les rubriques suivantes dans la référence de l'API AWS SDK pour Go .
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
/*
 *  Lambda function names appear as:
 *
 *  arn:aws:lambda:us-west-2:335556666777:function:HelloFunction
 *
 *  To find this value, look at the function in the AWS Management Console.
 *
 *  Before running this Java code example, set up your development environment,
 including your credentials.
```

```
 *
 *  For more information, see this documentation topic:
 *
 *  https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 *  This example performs the following tasks:
 *
 * 1. Creates an AWS Lambda function.
 * 2. Gets a specific AWS Lambda function.
 * 3. Lists all Lambda functions.
 * 4. Invokes a Lambda function.
 * 5. Updates the Lambda function code and invokes it again.
 * 6. Updates a Lambda function's configuration value.
 * 7. Deletes a Lambda function.
 */

public class LambdaScenario {
    public static final String DASHES = new String(new char[80]).replace("\0",
 "-");

    public static void main(String[] args) throws InterruptedException {
        final String usage = """

            Usage:
                <functionName> <role> <handler> <bucketName> <key>\s

            Where:
                functionName - The name of the Lambda function.\s
                role - The AWS Identity and Access Management (IAM) service role
 that has Lambda permissions.\s
                handler - The fully qualified method name (for example,
 example.Handler::handleRequest).\s
                bucketName - The Amazon Simple Storage Service (Amazon S3) bucket
 name that contains the .zip or .jar used to update the Lambda function's code.\s
                key - The Amazon S3 key name that represents the .zip or .jar
 (for example, LambdaHello-1.0-SNAPSHOT.jar).
                """;

        if (args.length != 5) {
              System.out.println(usage);
              return;
        }
```

```
        String functionName = args[0];
        String role = args[1];
        String handler = args[2];
        String bucketName = args[3];
        String key = args[4];
        LambdaClient awsLambda = LambdaClient.builder()
            .build();

        System.out.println(DASHES);
        System.out.println("Welcome to the AWS Lambda Basics scenario.");
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("1. Create an AWS Lambda function.");
        String funArn = createLambdaFunction(awsLambda, functionName, key,
bucketName, role, handler);
        System.out.println("The AWS Lambda ARN is " + funArn);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("2. Get the " + functionName + " AWS Lambda
function.");
        getFunction(awsLambda, functionName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("3. List all AWS Lambda functions.");
        listFunctions(awsLambda);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("4. Invoke the Lambda function.");
        System.out.println("*** Sleep for 1 min to get Lambda function ready.");
        Thread.sleep(60000);
        invokeFunction(awsLambda, functionName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("5. Update the Lambda function code and invoke it
again.");
        updateFunctionCode(awsLambda, functionName, bucketName, key);
        System.out.println("*** Sleep for 1 min to get Lambda function ready.");
        Thread.sleep(60000);
        invokeFunction(awsLambda, functionName);
```

```
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("6. Update a Lambda function's configuration value.");
        updateFunctionConfiguration(awsLambda, functionName, handler);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("7. Delete the AWS Lambda function.");
        LambdaScenario.deleteLambdaFunction(awsLambda, functionName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("The AWS Lambda scenario completed successfully");
        System.out.println(DASHES);
        awsLambda.close();
    }

    /**
     * Creates a new Lambda function in AWS using the AWS Lambda Java API.
     *
     * @param awsLambda    the AWS Lambda client used to interact with the AWS
Lambda service
     * @param functionName the name of the Lambda function to create
     * @param key          the S3 key of the function code
     * @param bucketName   the name of the S3 bucket containing the function code
     * @param role         the IAM role to assign to the Lambda function
     * @param handler      the fully qualified class name of the function handler
     * @return the Amazon Resource Name (ARN) of the created Lambda function
     */
    public static String createLambdaFunction(LambdaClient awsLambda,
                                               String functionName,
                                               String key,
                                               String bucketName,
                                               String role,
                                               String handler) {

        try {
            LambdaWaiter waiter = awsLambda.waiter();
            FunctionCode code = FunctionCode.builder()
                .s3Key(key)
                .s3Bucket(bucketName)
                .build();
```

```
            CreateFunctionRequest functionRequest =
CreateFunctionRequest.builder()
                .functionName(functionName)
                .description("Created by the Lambda Java API")
                .code(code)
                .handler(handler)
                .runtime(Runtime.JAVA17)
                .role(role)
                .build();

            // Create a Lambda function using a waiter
            CreateFunctionResponse functionResponse =
awsLambda.createFunction(functionRequest);
            GetFunctionRequest getFunctionRequest = GetFunctionRequest.builder()
                .functionName(functionName)
                .build();
            WaiterResponse<GetFunctionResponse> waiterResponse =
waiter.waitUntilFunctionExists(getFunctionRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            return functionResponse.functionArn();

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return "";
    }

    /**
     * Retrieves information about an AWS Lambda function.
     *
     * @param awsLambda    an instance of the {@link LambdaClient} class, which
is used to interact with the AWS Lambda service
     * @param functionName the name of the AWS Lambda function to retrieve
information about
     */
    public static void getFunction(LambdaClient awsLambda, String functionName) {
        try {
            GetFunctionRequest functionRequest = GetFunctionRequest.builder()
                .functionName(functionName)
                .build();

            GetFunctionResponse response =
awsLambda.getFunction(functionRequest);
```

```
            System.out.println("The runtime of this Lambda function is " +
response.configuration().runtime());

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Lists the AWS Lambda functions associated with the current AWS account.
     *
     * @param awsLambda an instance of the {@link LambdaClient} class, which is
used to interact with the AWS Lambda service
     *
     * @throws LambdaException if an error occurs while interacting with the AWS
Lambda service
     */
    public static void listFunctions(LambdaClient awsLambda) {
        try {
            ListFunctionsResponse functionResult = awsLambda.listFunctions();
            List<FunctionConfiguration> list = functionResult.functions();
            for (FunctionConfiguration config : list) {
                System.out.println("The function name is " +
config.functionName());
            }

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Invokes a specific AWS Lambda function.
     *
     * @param awsLambda    an instance of {@link LambdaClient} to interact with
the AWS Lambda service
     * @param functionName the name of the AWS Lambda function to be invoked
     */
    public static void invokeFunction(LambdaClient awsLambda, String
functionName) {
        InvokeResponse res;
        try {
```

```
            // Need a SdkBytes instance for the payload.
            JSONObject jsonObj = new JSONObject();
            jsonObj.put("inputValue", "2000");
            String json = jsonObj.toString();
            SdkBytes payload = SdkBytes.fromUtf8String(json);

            InvokeRequest request = InvokeRequest.builder()
                .functionName(functionName)
                .payload(payload)
                .build();

            res = awsLambda.invoke(request);
            String value = res.payload().asUtf8String();
            System.out.println(value);

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Updates the code for an AWS Lambda function.
     *
     * @param awsLambda  the AWS Lambda client
     * @param functionName the name of the Lambda function to update
     * @param bucketName the name of the S3 bucket where the function code is
located
     * @param key the key (file name) of the function code in the S3 bucket
     * @throws LambdaException if there is an error updating the function code
     */
    public static void updateFunctionCode(LambdaClient awsLambda, String
functionName, String bucketName, String key) {
        try {
            LambdaWaiter waiter = awsLambda.waiter();
            UpdateFunctionCodeRequest functionCodeRequest =
UpdateFunctionCodeRequest.builder()
                .functionName(functionName)
                .publish(true)
                .s3Bucket(bucketName)
                .s3Key(key)
                .build();
```

```
            UpdateFunctionCodeResponse response =
awsLambda.updateFunctionCode(functionCodeRequest);
            GetFunctionConfigurationRequest getFunctionConfigRequest =
GetFunctionConfigurationRequest.builder()
                .functionName(functionName)
                .build();

            WaiterResponse<GetFunctionConfigurationResponse> waiterResponse =
waiter
                .waitUntilFunctionUpdated(getFunctionConfigRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            System.out.println("The last modified value is " +
response.lastModified());

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Updates the configuration of an AWS Lambda function.
     *
     * @param awsLambda     the {@link LambdaClient} instance to use for the AWS
Lambda operation
     * @param functionName  the name of the AWS Lambda function to update
     * @param handler       the new handler for the AWS Lambda function
     *
     * @throws LambdaException if there is an error while updating the function
configuration
     */
    public static void updateFunctionConfiguration(LambdaClient awsLambda, String
functionName, String handler) {
        try {
            UpdateFunctionConfigurationRequest configurationRequest =
UpdateFunctionConfigurationRequest.builder()
                .functionName(functionName)
                .handler(handler)
                .runtime(Runtime.JAVA17)
                .build();

            awsLambda.updateFunctionConfiguration(configurationRequest);

        } catch (LambdaException e) {
```

```
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Deletes an AWS Lambda function.
     *
     * @param awsLambda     an instance of the {@link LambdaClient} class, which
 is used to interact with the AWS Lambda service
     * @param functionName  the name of the Lambda function to be deleted
     *
     * @throws LambdaException if an error occurs while deleting the Lambda
 function
     */
    public static void deleteLambdaFunction(LambdaClient awsLambda, String
 functionName) {
        try {
            DeleteFunctionRequest request = DeleteFunctionRequest.builder()
                .functionName(functionName)
                .build();

            awsLambda.deleteFunction(request);
            System.out.println("The " + functionName + " function was deleted");

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- Pour plus de détails sur l'API, consultez les rubriques suivantes dans la Référence des API du kit AWS SDK for Java 2.x .
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode

- UpdateFunctionConfiguration

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

Créez un rôle Gestion des identités et des accès AWS (IAM) qui accorde à Lambda l'autorisation d'écrire dans les journaux.

```
    logger.log(`Creating role (${NAME_ROLE_LAMBDA})...`);
    const response = await createRole(NAME_ROLE_LAMBDA);

import { AttachRolePolicyCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} policyArn
 * @param {string} roleName
 */
export const attachRolePolicy = (policyArn, roleName) => {
  const command = new AttachRolePolicyCommand({
    PolicyArn: policyArn,
    RoleName: roleName,
  });

  return client.send(command);
};
```

Créer une fonction Lambda et télécharger le code de gestionnaire.

```
const createFunction = async (funcName, roleArn) => {
  const client = new LambdaClient({});
```

```
  const code = await readFile(`${dirname}../functions/${funcName}.zip`);

  const command = new CreateFunctionCommand({
    Code: { ZipFile: code },
    FunctionName: funcName,
    Role: roleArn,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};
```

Invoquer la fonction avec un seul paramètre et obtenir des résultats.

```
const invoke = async (funcName, payload) => {
  const client = new LambdaClient({});
  const command = new InvokeCommand({
    FunctionName: funcName,
    Payload: JSON.stringify(payload),
    LogType: LogType.Tail,
  });

  const { Payload, LogResult } = await client.send(command);
  const result = Buffer.from(Payload).toString();
  const logs = Buffer.from(LogResult, "base64").toString();
  return { logs, result };
};
```

Mettre à jour le code de fonction et configurer son environnement Lambda avec une variable d'environnement.

```
const updateFunctionCode = async (funcName, newFunc) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${newFunc}.zip`);
  const command = new UpdateFunctionCodeCommand({
    ZipFile: code,
    FunctionName: funcName,
    Architectures: [Architecture.arm64],
```

```
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};

const updateFunctionConfiguration = (funcName) => {
  const client = new LambdaClient({});
  const config = readFileSync(`${dirname}../functions/config.json`).toString();
  const command = new UpdateFunctionConfigurationCommand({
    ...JSON.parse(config),
    FunctionName: funcName,
  });
  const result = client.send(command);
  waitForFunctionUpdated({ FunctionName: funcName });
  return result;
};
```

Répertorier les fonctions pour votre compte.

```
const listFunctions = () => {
  const client = new LambdaClient({});
  const command = new ListFunctionsCommand({});

  return client.send(command);
};
```

Supprimer le rôle IAM et la fonction Lambda.

```
import { DeleteRoleCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} roleName
 */
export const deleteRole = (roleName) => {
  const command = new DeleteRoleCommand({ RoleName: roleName });
```

```
  return client.send(command);
};

/**
 * @param {string} funcName
 */
const deleteFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new DeleteFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- Pour plus d'informations sur l'API, consultez les rubriques suivantes dans la référence de l'API AWS SDK pour JavaScript .
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Kotlin

SDK pour Kotlin

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
suspend fun main(args: Array<String>) {
    val usage = """
        Usage:
            <functionName> <role> <handler> <bucketName> <updatedBucketName>
 <key>
```

```
        Where:
            functionName - The name of the AWS Lambda function.
            role - The AWS Identity and Access Management (IAM) service role that
has AWS Lambda permissions.
            handler - The fully qualified method name (for example,
example.Handler::handleRequest).
            bucketName - The Amazon Simple Storage Service (Amazon S3) bucket
name that contains the ZIP or JAR used for the Lambda function's code.
            updatedBucketName - The Amazon S3 bucket name that contains the .zip
or .jar used to update the Lambda function's code.
            key - The Amazon S3 key name that represents the .zip or .jar file
(for example, LambdaHello-1.0-SNAPSHOT.jar).
            """

    if (args.size != 6) {
        println(usage)
        exitProcess(1)
    }

    val functionName = args[0]
    val role = args[1]
    val handler = args[2]
    val bucketName = args[3]
    val updatedBucketName = args[4]
    val key = args[5]

    println("Creating a Lambda function named $functionName.")
    val funArn = createScFunction(functionName, bucketName, key, handler, role)
    println("The AWS Lambda ARN is $funArn")

    // Get a specific Lambda function.
    println("Getting the $functionName AWS Lambda function.")
    getFunction(functionName)

    // List the Lambda functions.
    println("Listing all AWS Lambda functions.")
    listFunctionsSc()

    // Invoke the Lambda function.
    println("*** Invoke the Lambda function.")
    invokeFunctionSc(functionName)

    // Update the AWS Lambda function code.
```

```
    println("*** Update the Lambda function code.")
    updateFunctionCode(functionName, updatedBucketName, key)

    // println("*** Invoke the function again after updating the code.")
    invokeFunctionSc(functionName)

    // Update the AWS Lambda function configuration.
    println("Update the run time of the function.")
    updateFunctionConfiguration(functionName, handler)

    // Delete the AWS Lambda function.
    println("Delete the AWS Lambda function.")
    delFunction(functionName)
}

suspend fun createScFunction(
    myFunctionName: String,
    s3BucketName: String,
    myS3Key: String,
    myHandler: String,
    myRole: String,
): String {
    val functionCode =
        FunctionCode {
            s3Bucket = s3BucketName
            s3Key = myS3Key
        }

    val request =
        CreateFunctionRequest {
            functionName = myFunctionName
            code = functionCode
            description = "Created by the Lambda Kotlin API"
            handler = myHandler
            role = myRole
            runtime = Runtime.Java17
        }

    // Create a Lambda function using a waiter
    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        val functionResponse = awsLambda.createFunction(request)
        awsLambda.waitUntilFunctionActive {
            functionName = myFunctionName
        }
```

```
        return functionResponse.functionArn.toString()
    }
}

suspend fun getFunction(functionNameVal: String) {
    val functionRequest =
        GetFunctionRequest {
            functionName = functionNameVal
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        val response = awsLambda.getFunction(functionRequest)
        println("The runtime of this Lambda function is
 ${response.configuration?.runtime}")
    }
}

suspend fun listFunctionsSc() {
    val request =
        ListFunctionsRequest {
            maxItems = 10
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        val response = awsLambda.listFunctions(request)
        response.functions?.forEach { function ->
            println("The function name is ${function.functionName}")
        }
    }
}

suspend fun invokeFunctionSc(functionNameVal: String) {
    val json = """{"inputValue":"1000"}"""
    val byteArray = json.trimIndent().encodeToByteArray()
    val request =
        InvokeRequest {
            functionName = functionNameVal
            payload = byteArray
            logType = LogType.Tail
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        val res = awsLambda.invoke(request)
```

```
        println("The function payload is
 ${res.payload?.toString(Charsets.UTF_8)}")
    }
}

suspend fun updateFunctionCode(
    functionNameVal: String?,
    bucketName: String?,
    key: String?,
) {
    val functionCodeRequest =
        UpdateFunctionCodeRequest {
            functionName = functionNameVal
            publish = true
            s3Bucket = bucketName
            s3Key = key
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        val response = awsLambda.updateFunctionCode(functionCodeRequest)
        awsLambda.waitUntilFunctionUpdated {
            functionName = functionNameVal
        }
        println("The last modified value is " + response.lastModified)
    }
}

suspend fun updateFunctionConfiguration(
    functionNameVal: String?,
    handlerVal: String?,
) {
    val configurationRequest =
        UpdateFunctionConfigurationRequest {
            functionName = functionNameVal
            handler = handlerVal
            runtime = Runtime.Java17
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        awsLambda.updateFunctionConfiguration(configurationRequest)
    }
}

suspend fun delFunction(myFunctionName: String) {
```

```
    val request =
        DeleteFunctionRequest {
            functionName = myFunctionName
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        awsLambda.deleteFunction(request)
        println("$myFunctionName was deleted")
    }
}
```

- Pour plus de détails sur l'API, consultez les rubriques suivantes dans la Référence des API du kit AWS SDK pour Kotlin.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
namespace Lambda;

use Aws\S3\S3Client;
use GuzzleHttp\Psr7\Stream;
use Iam\IAMService;
```

```
class GettingStartedWithLambda
{
    public function run()
    {
        echo("\n");
        echo("---------------------------------------\n");
        print("Welcome to the AWS Lambda getting started demo using PHP!\n");
        echo("---------------------------------------\n");

        $clientArgs = [
            'region' => 'us-west-2',
            'version' => 'latest',
            'profile' => 'default',
        ];
        $uniqid = uniqid();

        $iamService = new IAMService();
        $s3client = new S3Client($clientArgs);
        $lambdaService = new LambdaService();

        echo "First, let's create a role to run our Lambda code.\n";
        $roleName = "test-lambda-role-$uniqid";
        $rolePolicyDocument = "{
            \"Version\": \"2012-10-17\",
            \"Statement\": [
                {
                    \"Effect\": \"Allow\",
                    \"Principal\": {
                        \"Service\": \"lambda.amazonaws.com\"
                    },
                    \"Action\": \"sts:AssumeRole\"
                }
            ]
        }";
        $role = $iamService->createRole($roleName, $rolePolicyDocument);
        echo "Created role {$role['RoleName']}.\n";

        $iamService->attachRolePolicy(
            $role['RoleName'],
            "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"
        );
        echo "Attached the AWSLambdaBasicExecutionRole to {$role['RoleName']}.
\n";
```

```
        echo "\nNow let's create an S3 bucket and upload our Lambda code there.
\n";
        $bucketName = "amzn-s3-demo-bucket-$uniqid";
        $s3client->createBucket([
            'Bucket' => $bucketName,
        ]);
        echo "Created bucket $bucketName.\n";

        $functionName = "doc_example_lambda_$uniqid";
        $codeBasic = __DIR__ . "/lambda_handler_basic.zip";
        $handler = "lambda_handler_basic";
        $file = file_get_contents($codeBasic);
        $s3client->putObject([
            'Bucket' => $bucketName,
            'Key' => $functionName,
            'Body' => $file,
        ]);
        echo "Uploaded the Lambda code.\n";

        $createLambdaFunction = $lambdaService->createFunction($functionName,
 $role, $bucketName, $handler);
        // Wait until the function has finished being created.
        do {
            $getLambdaFunction = $lambdaService-
>getFunction($createLambdaFunction['FunctionName']);
        } while ($getLambdaFunction['Configuration']['State'] == "Pending");
        echo "Created Lambda function {$getLambdaFunction['Configuration']
['FunctionName']}.\n";

        sleep(1);

        echo "\nOk, let's invoke that Lambda code.\n";
        $basicParams = [
            'action' => 'increment',
            'number' => 3,
        ];
        /** @var Stream $invokeFunction */
        $invokeFunction = $lambdaService->invoke($functionName, $basicParams)
['Payload'];
        $result = json_decode($invokeFunction->getContents())->result;
        echo "After invoking the Lambda code with the input of
 {$basicParams['number']} we received $result.\n";

        echo "\nSince that's working, let's update the Lambda code.\n";
```

```
        $codeCalculator = "lambda_handler_calculator.zip";
        $handlerCalculator = "lambda_handler_calculator";
        echo "First, put the new code into the S3 bucket.\n";
        $file = file_get_contents($codeCalculator);
        $s3client->putObject([
            'Bucket' => $bucketName,
            'Key' => $functionName,
            'Body' => $file,
        ]);
        echo "New code uploaded.\n";

        $lambdaService->updateFunctionCode($functionName, $bucketName,
 $functionName);
        // Wait for the Lambda code to finish updating.
        do {
            $getLambdaFunction = $lambdaService-
>getFunction($createLambdaFunction['FunctionName']);
        } while ($getLambdaFunction['Configuration']['LastUpdateStatus'] !==
 "Successful");
        echo "New Lambda code uploaded.\n";

        $environment = [
            'Variable' => ['Variables' => ['LOG_LEVEL' => 'DEBUG']],
        ];
        $lambdaService->updateFunctionConfiguration($functionName,
 $handlerCalculator, $environment);
        do {
            $getLambdaFunction = $lambdaService-
>getFunction($createLambdaFunction['FunctionName']);
        } while ($getLambdaFunction['Configuration']['LastUpdateStatus'] !==
 "Successful");
        echo "Lambda code updated with new handler and a LOG_LEVEL of DEBUG for
 more information.\n";

        echo "Invoke the new code with some new data.\n";
        $calculatorParams = [
            'action' => 'plus',
            'x' => 5,
            'y' => 4,
        ];
        $invokeFunction = $lambdaService->invoke($functionName,
 $calculatorParams, "Tail");
        $result = json_decode($invokeFunction['Payload']->getContents())->result;
```

```
        echo "Indeed, {$calculatorParams['x']} + {$calculatorParams['y']} does
equal $result.\n";
        echo "Here's the extra debug info: ";
        echo base64_decode($invokeFunction['LogResult']) . "\n";

        echo "\nBut what happens if you try to divide by zero?\n";
        $divZeroParams = [
            'action' => 'divide',
            'x' => 5,
            'y' => 0,
        ];
        $invokeFunction = $lambdaService->invoke($functionName, $divZeroParams,
"Tail");
        $result = json_decode($invokeFunction['Payload']->getContents())->result;
        echo "You get a |$result| result.\n";
        echo "And an error message: ";
        echo base64_decode($invokeFunction['LogResult']) . "\n";

        echo "\nHere's all the Lambda functions you have in this Region:\n";
        $listLambdaFunctions = $lambdaService->listFunctions(5);
        $allLambdaFunctions = $listLambdaFunctions['Functions'];
        $next = $listLambdaFunctions->get('NextMarker');
        while ($next != false) {
            $listLambdaFunctions = $lambdaService->listFunctions(5, $next);
            $next = $listLambdaFunctions->get('NextMarker');
            $allLambdaFunctions = array_merge($allLambdaFunctions,
$listLambdaFunctions['Functions']);
        }
        foreach ($allLambdaFunctions as $function) {
            echo "{$function['FunctionName']}\n";
        }

        echo "\n\nAnd don't forget to clean up your data!\n";

        $lambdaService->deleteFunction($functionName);
        echo "Deleted Lambda function.\n";
        $iamService->deleteRole($role['RoleName']);
        echo "Deleted Role.\n";
        $deleteObjects = $s3client->listObjectsV2([
            'Bucket' => $bucketName,
        ]);
        $deleteObjects = $s3client->deleteObjects([
            'Bucket' => $bucketName,
            'Delete' => [
```

```
                'Objects' => $deleteObjects['Contents'],
            ]
        ]);
        echo "Deleted all objects from the S3 bucket.\n";
        $s3client->deleteBucket(['Bucket' => $bucketName]);
        echo "Deleted the bucket.\n";
    }
}
```

- Pour plus de détails sur l'API, consultez les rubriques suivantes dans la Référence des API du kit AWS SDK pour PHP .
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

Définir un gestionnaire Lambda qui incrémente un nombre.

```
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)
```

```
def lambda_handler(event, context):
    """
    Accepts an action and a single number, performs the specified action on the
 number,
    and returns the result. The only allowable action is 'increment'.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the action.
    """
    result = None
    action = event.get("action")
    if action == "increment":
        result = event.get("number", 0) + 1
        logger.info("Calculated result of %s", result)
    else:
        logger.error("%s is not a valid action.", action)

    response = {"result": result}
    return response
```

Définir un deuxième gestionnaire Lambda qui effectue des opérations arithmétiques.

```
import logging
import os


logger = logging.getLogger()

# Define a list of Python lambda functions that are called by this AWS Lambda
 function.
ACTIONS = {
    "plus": lambda x, y: x + y,
    "minus": lambda x, y: x - y,
    "times": lambda x, y: x * y,
    "divided-by": lambda x, y: x / y,
}
```

```
def lambda_handler(event, context):
    """
    Accepts an action and two numbers, performs the specified action on the
 numbers,
    and returns the result.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the specified action.
    """
    # Set the log level based on a variable configured in the Lambda environment.
    logger.setLevel(os.environ.get("LOG_LEVEL", logging.INFO))
    logger.debug("Event: %s", event)

    action = event.get("action")
    func = ACTIONS.get(action)
    x = event.get("x")
    y = event.get("y")
    result = None
    try:
        if func is not None and x is not None and y is not None:
            result = func(x, y)
            logger.info("%s %s %s is %s", x, action, y, result)
        else:
            logger.error("I can't calculate %s %s %s.", x, action, y)
    except ZeroDivisionError:
        logger.warning("I can't divide %s by 0!", x)

    response = {"result": result}
    return response
```

Créer des fonctions qui enveloppent les actions Lambda.

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource
```

```
    @staticmethod
    def create_deployment_package(source_file, destination_file):
        """
        Creates a Lambda deployment package in .zip format in an in-memory
buffer. This
        buffer can be passed directly to Lambda when creating the function.

        :param source_file: The name of the file that contains the Lambda handler
                            function.
        :param destination_file: The name to give the file when it's deployed to
Lambda.
        :return: The deployment package.
        """
        buffer = io.BytesIO()
        with zipfile.ZipFile(buffer, "w") as zipped:
            zipped.write(source_file, destination_file)
        buffer.seek(0)
        return buffer.read()

    def get_iam_role(self, iam_role_name):
        """
        Get an AWS Identity and Access Management (IAM) role.

        :param iam_role_name: The name of the role to retrieve.
        :return: The IAM role.
        """
        role = None
        try:
            temp_role = self.iam_resource.Role(iam_role_name)
            temp_role.load()
            role = temp_role
            logger.info("Got IAM role %s", role.name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "NoSuchEntity":
                logger.info("IAM role %s does not exist.", iam_role_name)
            else:
                logger.error(
                    "Couldn't get IAM role %s. Here's why: %s: %s",
                    iam_role_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
```

```
                raise
        return role

    def create_iam_role_for_lambda(self, iam_role_name):
        """
        Creates an IAM role that grants the Lambda function basic permissions. If
a
        role with the specified name already exists, it is used for the demo.

        :param iam_role_name: The name of the role to create.
        :return: The role and a value that indicates whether the role is newly
created.
        """
        role = self.get_iam_role(iam_role_name)
        if role is not None:
            return role, False

        lambda_assume_role_policy = {
            "Version":"2012-10-17",
            "Statement": [
                {
                    "Effect": "Allow",
                    "Principal": {"Service": "lambda.amazonaws.com"},
                    "Action": "sts:AssumeRole",
                }
            ],
        }
        policy_arn = "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole"

        try:
            role = self.iam_resource.create_role(
                RoleName=iam_role_name,
                AssumeRolePolicyDocument=json.dumps(lambda_assume_role_policy),
            )
            logger.info("Created role %s.", role.name)
            role.attach_policy(PolicyArn=policy_arn)
            logger.info("Attached basic execution policy to role %s.", role.name)
        except ClientError as error:
            if error.response["Error"]["Code"] == "EntityAlreadyExists":
                role = self.iam_resource.Role(iam_role_name)
                logger.warning("The role %s already exists. Using it.",
iam_role_name)
            else:
```

```
                logger.exception(
                    "Couldn't create role %s or attach policy %s.",
                    iam_role_name,
                    policy_arn,
                )
                raise

        return role, True

    def get_function(self, function_name):
        """
        Gets data about a Lambda function.

        :param function_name: The name of the function.
        :return: The function data.
        """
        response = None
        try:
            response =
self.lambda_client.get_function(FunctionName=function_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.info("Function %s does not exist.", function_name)
            else:
                logger.error(
                    "Couldn't get function %s. Here's why: %s: %s",
                    function_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        return response


    def create_function(
        self, function_name, handler_name, iam_role, deployment_package
    ):
        """
        Deploys a Lambda function.

        :param function_name: The name of the Lambda function.
        :param handler_name: The fully qualified name of the handler function.
This
                             must include the file name and the function name.
```

```
        :param iam_role: The IAM role to use for the function.
        :param deployment_package: The deployment package that contains the
function
                                   code in .zip format.
        :return: The Amazon Resource Name (ARN) of the newly created function.
        """
        try:
            response = self.lambda_client.create_function(
                FunctionName=function_name,
                Description="AWS Lambda doc example",
                Runtime="python3.9",
                Role=iam_role.arn,
                Handler=handler_name,
                Code={"ZipFile": deployment_package},
                Publish=True,
            )
            function_arn = response["FunctionArn"]
            waiter = self.lambda_client.get_waiter("function_active_v2")
            waiter.wait(FunctionName=function_name)
            logger.info(
                "Created function '%s' with ARN: '%s'.",
                function_name,
                response["FunctionArn"],
            )
        except ClientError:
            logger.error("Couldn't create function %s.", function_name)
            raise
        else:
            return function_arn


    def delete_function(self, function_name):
        """
        Deletes a Lambda function.

        :param function_name: The name of the function to delete.
        """
        try:
            self.lambda_client.delete_function(FunctionName=function_name)
        except ClientError:
            logger.exception("Couldn't delete function %s.", function_name)
            raise
```

```
    def invoke_function(self, function_name, function_params, get_log=False):
        """
        Invokes a Lambda function.

        :param function_name: The name of the function to invoke.
        :param function_params: The parameters of the function as a dict. This
dict
                                is serialized to JSON before it is sent to
Lambda.
        :param get_log: When true, the last 4 KB of the execution log are
included in
                        the response.
        :return: The response from the function invocation.
        """
        try:
            response = self.lambda_client.invoke(
                FunctionName=function_name,
                Payload=json.dumps(function_params),
                LogType="Tail" if get_log else "None",
            )
            logger.info("Invoked function %s.", function_name)
        except ClientError:
            logger.exception("Couldn't invoke function %s.", function_name)
            raise
        return response


    def update_function_code(self, function_name, deployment_package):
        """
        Updates the code for a Lambda function by submitting a .zip archive that
contains
        the code for the function.

        :param function_name: The name of the function to update.
        :param deployment_package: The function code to update, packaged as bytes
in
                                   .zip format.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_code(
                FunctionName=function_name, ZipFile=deployment_package
            )
        except ClientError as err:
```

```
            logger.error(
                "Couldn't update function %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response


    def update_function_configuration(self, function_name, env_vars):
        """
        Updates the environment variables for a Lambda function.

        :param function_name: The name of the function to update.
        :param env_vars: A dict of environment variables to update.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_configuration(
                FunctionName=function_name, Environment={"Variables": env_vars}
            )
        except ClientError as err:
            logger.error(
                "Couldn't update function configuration %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response


    def list_functions(self):
        """
        Lists the Lambda functions for the current account.
        """
        try:
            func_paginator = self.lambda_client.get_paginator("list_functions")
            for func_page in func_paginator.paginate():
                for func in func_page["Functions"]:
                    print(func["FunctionName"])
```

```
                    desc = func.get("Description")
                    if desc:
                        print(f"\t{desc}")
                    print(f"\t{func['Runtime']}: {func['Handler']}")
        except ClientError as err:
            logger.error(
                "Couldn't list functions. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

Créer une fonction qui exécute le scénario.

```
class UpdateFunctionWaiter(CustomWaiter):
    """A custom waiter that waits until a function is successfully updated."""

    def __init__(self, client):
        super().__init__(
            "UpdateSuccess",
            "GetFunction",
            "Configuration.LastUpdateStatus",
            {"Successful": WaitState.SUCCESS, "Failed": WaitState.FAILURE},
            client,
        )

    def wait(self, function_name):
        self._wait(FunctionName=function_name)


def run_scenario(lambda_client, iam_resource, basic_file, calculator_file,
 lambda_name):
    """
    Runs the scenario.

    :param lambda_client: A Boto3 Lambda client.
    :param iam_resource: A Boto3 IAM resource.
    :param basic_file: The name of the file that contains the basic Lambda
 handler.
```

```
    :param calculator_file: The name of the file that contains the calculator
Lambda handler.
    :param lambda_name: The name to give resources created for the scenario, such
as the
                        IAM role and the Lambda function.
    """
    logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")

    print("-" * 88)
    print("Welcome to the AWS Lambda getting started with functions demo.")
    print("-" * 88)

    wrapper = LambdaWrapper(lambda_client, iam_resource)

    print("Checking for IAM role for Lambda...")
    iam_role, should_wait = wrapper.create_iam_role_for_lambda(lambda_name)
    if should_wait:
        logger.info("Giving AWS time to create resources...")
        wait(10)

    print(f"Looking for function {lambda_name}...")
    function = wrapper.get_function(lambda_name)
    if function is None:
        print("Zipping the Python script into a deployment package...")
        deployment_package = wrapper.create_deployment_package(
            basic_file, f"{lambda_name}.py"
        )
        print(f"...and creating the {lambda_name} Lambda function.")
        wrapper.create_function(
            lambda_name, f"{lambda_name}.lambda_handler", iam_role,
deployment_package
        )
    else:
        print(f"Function {lambda_name} already exists.")
    print("-" * 88)

    print(f"Let's invoke {lambda_name}. This function increments a number.")
    action_params = {
        "action": "increment",
        "number": q.ask("Give me a number to increment: ", q.is_int),
    }
    print(f"Invoking {lambda_name}...")
    response = wrapper.invoke_function(lambda_name, action_params)
    print(
```

```
        f"Incrementing {action_params['number']} resulted in "
        f"{json.load(response['Payload'])}"
    )
    print("-" * 88)

    print(f"Let's update the function to an arithmetic calculator.")
    q.ask("Press Enter when you're ready.")
    print("Creating a new deployment package...")
    deployment_package = wrapper.create_deployment_package(
        calculator_file, f"{lambda_name}.py"
    )
    print(f"...and updating the {lambda_name} Lambda function.")
    update_waiter = UpdateFunctionWaiter(lambda_client)
    wrapper.update_function_code(lambda_name, deployment_package)
    update_waiter.wait(lambda_name)
    print(f"This function uses an environment variable to control logging
level.")
    print(f"Let's set it to DEBUG to get the most logging.")
    wrapper.update_function_configuration(
        lambda_name, {"LOG_LEVEL": logging.getLevelName(logging.DEBUG)}
    )

    actions = ["plus", "minus", "times", "divided-by"]
    want_invoke = True
    while want_invoke:
        print(f"Let's invoke {lambda_name}. You can invoke these actions:")
        for index, action in enumerate(actions):
            print(f"{index + 1}: {action}")
        action_params = {}
        action_index = q.ask(
            "Enter the number of the action you want to take: ",
            q.is_int,
            q.in_range(1, len(actions)),
        )
        action_params["action"] = actions[action_index - 1]
        print(f"You've chosen to invoke 'x {action_params['action']} y'.")
        action_params["x"] = q.ask("Enter a value for x: ", q.is_int)
        action_params["y"] = q.ask("Enter a value for y: ", q.is_int)
        print(f"Invoking {lambda_name}...")
        response = wrapper.invoke_function(lambda_name, action_params, True)
        print(
            f"Calculating {action_params['x']} {action_params['action']}
{action_params['y']} "
            f"resulted in {json.load(response['Payload'])}"
```

```
        )
        q.ask("Press Enter to see the logs from the call.")
        print(base64.b64decode(response["LogResult"]).decode())
        want_invoke = q.ask("That was fun. Shall we do it again? (y/n) ",
 q.is_yesno)
    print("-" * 88)

    if q.ask(
        "Do you want to list all of the functions in your account? (y/n) ",
 q.is_yesno
    ):
        wrapper.list_functions()
    print("-" * 88)

    if q.ask("Ready to delete the function and role? (y/n) ", q.is_yesno):
        for policy in iam_role.attached_policies.all():
            policy.detach_role(RoleName=iam_role.name)
        iam_role.delete()
        print(f"Deleted role {lambda_name}.")
        wrapper.delete_function(lambda_name)
        print(f"Deleted function {lambda_name}.")

    print("\nThanks for watching!")
    print("-" * 88)


if __name__ == "__main__":
    try:
        run_scenario(
            boto3.client("lambda"),
            boto3.resource("iam"),
            "lambda_handler_basic.py",
            "lambda_handler_calculator.py",
            "doc_example_lambda_calculator",
        )
    except Exception:
        logging.exception("Something went wrong with the demo!")
```

- Pour plus d'informations sur l'API, consultez les rubriques suivantes dans AWS SDK for Python (Boto3) API Reference.
  - CreateFunction

- DeleteFunction
- GetFunction
- Invoke
- ListFunctions
- UpdateFunctionCode
- UpdateFunctionConfiguration

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

Configurez les autorisations IAM préalables pour une fonction Lambda capable d'écrire des journaux.

```
# Get an AWS Identity and Access Management (IAM) role.
#
# @param iam_role_name: The name of the role to retrieve.
# @param action: Whether to create or destroy the IAM apparatus.
# @return: The IAM role.
def manage_iam(iam_role_name, action)
  case action
  when 'create'
    create_iam_role(iam_role_name)
  when 'destroy'
    destroy_iam_role(iam_role_name)
  else
    raise "Incorrect action provided. Must provide 'create' or 'destroy'"
  end
end

private

def create_iam_role(iam_role_name)
```

```
    role_policy = {
      'Version': '2012-10-17',
      'Statement': [
        {
          'Effect': 'Allow',
          'Principal': { 'Service': 'lambda.amazonaws.com' },
          'Action': 'sts:AssumeRole'
        }
      ]
    }
    role = @iam_client.create_role(
      role_name: iam_role_name,
      assume_role_policy_document: role_policy.to_json
    )
    @iam_client.attach_role_policy(
      {
        policy_arn: 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole',
        role_name: iam_role_name
      }
    )
    wait_for_role_to_exist(iam_role_name)
    @logger.debug("Successfully created IAM role: #{role['role']['arn']}")
    sleep(10)
    [role, role_policy.to_json]
  end

  def destroy_iam_role(iam_role_name)
    @iam_client.detach_role_policy(
      {
        policy_arn: 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole',
        role_name: iam_role_name
      }
    )
    @iam_client.delete_role(role_name: iam_role_name)
    @logger.debug("Detached policy & deleted IAM role: #{iam_role_name}")
  end

  def wait_for_role_to_exist(iam_role_name)
    @iam_client.wait_until(:role_exists, { role_name: iam_role_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
```

```
  end
```

Définissez un gestionnaire Lambda qui incrémente un nombre fourni en tant que paramètre d'invocation.

```
require 'logger'

# A function that increments a whole number by one (1) and logs the result.
# Requires a manually-provided runtime parameter, 'number', which must be Int
#
# @param event [Hash] Parameters sent when the function is invoked
# @param context [Hash] Methods and properties that provide information
# about the invocation, function, and execution environment.
# @return incremented_number [String] The incremented number.
def lambda_handler(event:, context:)
  logger = Logger.new($stdout)
  log_level = ENV['LOG_LEVEL']
  logger.level = case log_level
                 when 'debug'
                   Logger::DEBUG
                 when 'info'
                   Logger::INFO
                 else
                   Logger::ERROR
                 end
  logger.debug('This is a debug log message.')
  logger.info('This is an info log message. Code executed successfully!')
  number = event['number'].to_i
  incremented_number = number + 1
  logger.info("You provided #{number.round} and it was incremented to
 #{incremented_number.round}")
  incremented_number.round.to_s
end
```

Zippez votre fonction Lambda dans un package de déploiement.

```
  # Creates a Lambda deployment package in .zip format.
  #
  # @param source_file: The name of the object, without suffix, for the Lambda
 file and zip.
  # @return: The deployment package.
```

```
 def create_deployment_package(source_file)
   Dir.chdir(File.dirname(__FILE__))
   if File.exist?('lambda_function.zip')
     File.delete('lambda_function.zip')
     @logger.debug('Deleting old zip: lambda_function.zip')
   end
   Zip::File.open('lambda_function.zip', create: true) do |zipfile|
     zipfile.add('lambda_function.rb', "#{source_file}.rb")
   end
   @logger.debug("Zipping #{source_file}.rb into: lambda_function.zip.")
   File.read('lambda_function.zip').to_s
 rescue StandardError => e
   @logger.error("There was an error creating deployment package:\n
#{e.message}")
 end
```

Créez une fonction Lambda.

```
 # Deploys a Lambda function.
 #
 # @param function_name: The name of the Lambda function.
 # @param handler_name: The fully qualified name of the handler function.
 # @param role_arn: The IAM role to use for the function.
 # @param deployment_package: The deployment package that contains the function
code in .zip format.
 # @return: The Amazon Resource Name (ARN) of the newly created function.
 def create_function(function_name, handler_name, role_arn, deployment_package)
   response = @lambda_client.create_function({
                                               role: role_arn.to_s,
                                               function_name: function_name,
                                               handler: handler_name,
                                               runtime: 'ruby2.7',
                                               code: {
                                                 zip_file: deployment_package
                                               },
                                               environment: {
                                                 variables: {
                                                   'LOG_LEVEL' => 'info'
                                                 }
                                               }
                                             })
```

```
    @lambda_client.wait_until(:function_active_v2, { function_name:
function_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
    response
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error creating #{function_name}:\n #{e.message}")
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
  end
```

Invoquez votre fonction Lambda avec des paramètres d'exécution facultatifs.

```
  # Invokes a Lambda function.
  # @param function_name [String] The name of the function to invoke.
  # @param payload [nil] Payload containing runtime parameters.
  # @return [Object] The response from the function invocation.
  def invoke_function(function_name, payload = nil)
    params = { function_name: function_name }
    params[:payload] = payload unless payload.nil?
    @lambda_client.invoke(params)
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error executing #{function_name}:\n
#{e.message}")
  end
```

Mettez la configuration de votre fonction Lambda à jour pour injecter une nouvelle variable d'environnement.

```
  # Updates the environment variables for a Lambda function.
  # @param function_name: The name of the function to update.
  # @param log_level: The log level of the function.
  # @return: Data about the update, including the status.
  def update_function_configuration(function_name, log_level)
    @lambda_client.update_function_configuration({
                                                   function_name: function_name,
                                                   environment: {
                                                     variables: {
                                                       'LOG_LEVEL' => log_level
```

```
                                                    }
                                                  }
                                                })
   @lambda_client.wait_until(:function_updated_v2, { function_name:
function_name }) do |w|
     w.max_attempts = 5
     w.delay = 5
   end
 rescue Aws::Lambda::Errors::ServiceException => e
   @logger.error("There was an error updating configurations for
#{function_name}:\n #{e.message}")
 rescue Aws::Waiters::Errors::WaiterFailed => e
   @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
 end
```

Mettez le code de votre fonction Lambda à jour avec un package de déploiement différent contenant un code différent.

```
 # Updates the code for a Lambda function by submitting a .zip archive that
contains
 # the code for the function.
 #
 # @param function_name: The name of the function to update.
 # @param deployment_package: The function code to update, packaged as bytes in
 #                            .zip format.
 # @return: Data about the update, including the status.
 def update_function_code(function_name, deployment_package)
   @lambda_client.update_function_code(
     function_name: function_name,
     zip_file: deployment_package
   )
   @lambda_client.wait_until(:function_updated_v2, { function_name:
function_name }) do |w|
     w.max_attempts = 5
     w.delay = 5
   end
 rescue Aws::Lambda::Errors::ServiceException => e
   @logger.error("There was an error updating function code for:
#{function_name}:\n #{e.message}")
   nil
 rescue Aws::Waiters::Errors::WaiterFailed => e
```

```
    @logger.error("Failed waiting for #{function_name} to update:\n
#{e.message}")
  end
```

Répertoriez toutes les fonctions Lambda existantes à l'aide du programme de pagination intégré.

```
# Lists the Lambda functions for the current account.
def list_functions
  functions = []
  @lambda_client.list_functions.each do |response|
    response['functions'].each do |function|
      functions.append(function['function_name'])
    end
  end
  functions
rescue Aws::Lambda::Errors::ServiceException => e
  @logger.error("There was an error listing functions:\n #{e.message}")
end
```

Supprimez une fonction Lambda spécifique.

```
# Deletes a Lambda function.
# @param function_name: The name of the function to delete.
def delete_function(function_name)
  print "Deleting function: #{function_name}..."
  @lambda_client.delete_function(
    function_name: function_name
  )
  print 'Done!'.green
rescue Aws::Lambda::Errors::ServiceException => e
  @logger.error("There was an error deleting #{function_name}:\n #{e.message}")
end
```

- Pour plus d'informations sur l'API, consultez les rubriques suivantes dans la référence de l'API AWS SDK pour Ruby .
  - CreateFunction
  - DeleteFunction

- GetFunction
- Invoke
- ListFunctions
- UpdateFunctionCode
- UpdateFunctionConfiguration

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

Le fichier Cargo.toml avec les dépendances utilisées dans ce scénario.

```
[package]
name = "lambda-code-examples"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/
reference/manifest.html

[dependencies]
aws-config = { version = "1.0.1", features = ["behavior-version-latest"] }
aws-sdk-ec2 = { version = "1.3.0" }
aws-sdk-iam = { version = "1.3.0" }
aws-sdk-lambda = { version = "1.3.0" }
aws-sdk-s3 = { version = "1.4.0" }
aws-smithy-types = { version = "1.0.1" }
aws-types = { version = "1.0.1" }
clap = { version = "4.4", features = ["derive"] }
tokio = { version = "1.20.1", features = ["full"] }
tracing-subscriber = { version = "0.3.15", features = ["env-filter"] }
tracing = "0.1.37"
serde_json = "1.0.94"
anyhow = "1.0.71"
```

```
uuid = { version = "1.3.3", features = ["v4"] }
lambda_runtime = "0.8.0"
serde = "1.0.164"
```

Un ensemble d'utilitaires qui simplifient l'appel à Lambda pour ce scénario. Ce fichier est src/ations.rs dans la caisse.

```
use anyhow::anyhow;
use aws_sdk_iam::operation::{create_role::CreateRoleError,
 delete_role::DeleteRoleOutput};
use aws_sdk_lambda::{
    operation::{
        delete_function::DeleteFunctionOutput, get_function::GetFunctionOutput,
        invoke::InvokeOutput, list_functions::ListFunctionsOutput,
        update_function_code::UpdateFunctionCodeOutput,
        update_function_configuration::UpdateFunctionConfigurationOutput,
    },
    primitives::ByteStream,
    types::{Environment, FunctionCode, LastUpdateStatus, State},
};
use aws_sdk_s3::{
    error::ErrorMetadata,
    operation::{delete_bucket::DeleteBucketOutput,
 delete_object::DeleteObjectOutput},
    types::CreateBucketConfiguration,
};
use aws_smithy_types::Blob;
use serde::{ser::SerializeMap, Serialize};
use std::{fmt::Display, path::PathBuf, str::FromStr, time::Duration};
use tracing::{debug, info, warn};

/* Operation describes  */
#[derive(Clone, Copy, Debug, Serialize)]
pub enum Operation {
    #[serde(rename = "plus")]
    Plus,
    #[serde(rename = "minus")]
    Minus,
    #[serde(rename = "times")]
    Times,
    #[serde(rename = "divided-by")]
```

```
    DividedBy,
}

impl FromStr for Operation {
    type Err = anyhow::Error;

    fn from_str(s: &str) -> Result<Self, Self::Err> {
        match s {
            "plus" => Ok(Operation::Plus),
            "minus" => Ok(Operation::Minus),
            "times" => Ok(Operation::Times),
            "divided-by" => Ok(Operation::DividedBy),
            _ => Err(anyhow!("Unknown operation {s}")),
        }
    }
}

impl Display for Operation {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            Operation::Plus => write!(f, "plus"),
            Operation::Minus => write!(f, "minus"),
            Operation::Times => write!(f, "times"),
            Operation::DividedBy => write!(f, "divided-by"),
        }
    }
}

/**
 * InvokeArgs will be serialized as JSON and sent to the AWS Lambda handler.
 */
#[derive(Debug)]
pub enum InvokeArgs {
    Increment(i32),
    Arithmetic(Operation, i32, i32),
}

impl Serialize for InvokeArgs {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer,
    {
        match self {
            InvokeArgs::Increment(i) => serializer.serialize_i32(*i),
```

```
            InvokeArgs::Arithmetic(o, i, j) => {
                let mut map: S::SerializeMap =
 serializer.serialize_map(Some(3))?;
                map.serialize_key(&"op".to_string())?;
                map.serialize_value(&o.to_string())?;
                map.serialize_key(&"i".to_string())?;
                map.serialize_value(&i)?;
                map.serialize_key(&"j".to_string())?;
                map.serialize_value(&j)?;
                map.end()
            }
        }
    }
}

/** A policy document allowing Lambda to execute this function on the account's
 behalf. */
const ROLE_POLICY_DOCUMENT: &str = r#"{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": { "Service": "lambda.amazonaws.com" },
            "Action": "sts:AssumeRole"
        }
    ]
}"#;

/**
 * A LambdaManager gathers all the resources necessary to run the Lambda example
 scenario.
 * This includes instantiated aws_sdk clients and details of resource names.
 */
pub struct LambdaManager {
    iam_client: aws_sdk_iam::Client,
    lambda_client: aws_sdk_lambda::Client,
    s3_client: aws_sdk_s3::Client,
    lambda_name: String,
    role_name: String,
    bucket: String,
    own_bucket: bool,
}
```

```
// These unit type structs provide nominal typing on top of String parameters for
 LambdaManager::new
pub struct LambdaName(pub String);
pub struct RoleName(pub String);
pub struct Bucket(pub String);
pub struct OwnBucket(pub bool);

impl LambdaManager {
    pub fn new(
        iam_client: aws_sdk_iam::Client,
        lambda_client: aws_sdk_lambda::Client,
        s3_client: aws_sdk_s3::Client,
        lambda_name: LambdaName,
        role_name: RoleName,
        bucket: Bucket,
        own_bucket: OwnBucket,
    ) -> Self {
        Self {
            iam_client,
            lambda_client,
            s3_client,
            lambda_name: lambda_name.0,
            role_name: role_name.0,
            bucket: bucket.0,
            own_bucket: own_bucket.0,
        }
    }

    /**
     * Load the AWS configuration from the environment.
     * Look up lambda_name and bucket if none are given, or generate a random
 name if not present in the environment.
     * If the bucket name is provided, the caller needs to have created the
 bucket.
     * If the bucket name is generated, it will be created.
     */
    pub async fn load_from_env(lambda_name: Option<String>, bucket:
 Option<String>) -> Self {
        let sdk_config = aws_config::load_from_env().await;
        let lambda_name = LambdaName(lambda_name.unwrap_or_else(|| {
            std::env::var("LAMBDA_NAME").unwrap_or_else(|_|
 "rust_lambda_example".to_string())
        }));
        let role_name = RoleName(format!("{}_role", lambda_name.0));
```

```
        let (bucket, own_bucket) =
            match bucket {
                Some(bucket) => (Bucket(bucket), false),
                None => (
                    Bucket(std::env::var("LAMBDA_BUCKET").unwrap_or_else(|_| {
                        format!("rust-lambda-example-{}", uuid::Uuid::new_v4())
                    })),
                    true,
                ),
            };

        let s3_client = aws_sdk_s3::Client::new(&sdk_config);

        if own_bucket {
            info!("Creating bucket for demo: {}", bucket.0);
            s3_client
                .create_bucket()
                .bucket(bucket.0.clone())
                .create_bucket_configuration(
                    CreateBucketConfiguration::builder()

.location_constraint(aws_sdk_s3::types::BucketLocationConstraint::from(
                            sdk_config.region().unwrap().as_ref(),
                        ))
                        .build(),
                )
                .send()
                .await
                .unwrap();
        }

        Self::new(
            aws_sdk_iam::Client::new(&sdk_config),
            aws_sdk_lambda::Client::new(&sdk_config),
            s3_client,
            lambda_name,
            role_name,
            bucket,
            OwnBucket(own_bucket),
        )
    }

    /**
     * Upload function code from a path to a zip file.
```

```
     * The zip file must have an AL2 Linux-compatible binary called `bootstrap`.
     * The easiest way to create such a zip is to use `cargo lambda build --
output-format Zip`.
     */
    async fn prepare_function(
        &self,
        zip_file: PathBuf,
        key: Option<String>,
    ) -> Result<FunctionCode, anyhow::Error> {
        let body = ByteStream::from_path(zip_file).await?;

        let key = key.unwrap_or_else(|| format!("{}_code", self.lambda_name));

        info!("Uploading function code to s3://{}/{}", self.bucket, key);
        let _ = self
            .s3_client
            .put_object()
            .bucket(self.bucket.clone())
            .key(key.clone())
            .body(body)
            .send()
            .await?;

        Ok(FunctionCode::builder()
            .s3_bucket(self.bucket.clone())
            .s3_key(key)
            .build())
    }

    /**
     * Create a function, uploading from a zip file.
     */
    pub async fn create_function(&self, zip_file: PathBuf) -> Result<String,
 anyhow::Error> {
        let code = self.prepare_function(zip_file, None).await?;

        let key = code.s3_key().unwrap().to_string();

        let role = self.create_role().await.map_err(|e| anyhow!(e))?;

        info!("Created iam role, waiting 15s for it to become active");
        tokio::time::sleep(Duration::from_secs(15)).await;

        info!("Creating lambda function {}", self.lambda_name);
```

```
        let _ = self
            .lambda_client
            .create_function()
            .function_name(self.lambda_name.clone())
            .code(code)
            .role(role.arn())
            .runtime(aws_sdk_lambda::types::Runtime::Providedal2)
            .handler("_unused")
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        self.lambda_client
            .publish_version()
            .function_name(self.lambda_name.clone())
            .send()
            .await?;

        Ok(key)
    }

    /**
     * Create an IAM execution role for the managed Lambda function.
     * If the role already exists, use that instead.
     */
    async fn create_role(&self) -> Result<aws_sdk_iam::types::Role,
CreateRoleError> {
        info!("Creating execution role for function");
        let get_role = self
            .iam_client
            .get_role()
            .role_name(self.role_name.clone())
            .send()
            .await;
        if let Ok(get_role) = get_role {
            if let Some(role) = get_role.role {
                return Ok(role);
            }
        }

        let create_role = self
            .iam_client
```

```
            .create_role()
            .role_name(self.role_name.clone())
            .assume_role_policy_document(ROLE_POLICY_DOCUMENT)
            .send()
            .await;

        match create_role {
            Ok(create_role) => match create_role.role {
                Some(role) => Ok(role),
                None => Err(CreateRoleError::generic(
                    ErrorMetadata::builder()
                        .message("CreateRole returned empty success")
                        .build(),
                )),
            },
            Err(err) => Err(err.into_service_error()),
        }
    }

    /**
     * Poll `is_function_ready` with a 1-second delay. It returns when the
function is ready or when there's an error checking the function's state.
     */
    pub async fn wait_for_function_ready(&self) -> Result<(), anyhow::Error> {
        info!("Waiting for function");
        while !self.is_function_ready(None).await? {
            info!("Function is not ready, sleeping 1s");
            tokio::time::sleep(Duration::from_secs(1)).await;
        }
        Ok(())
    }

    /**
     * Check if a Lambda function is ready to be invoked.
     * A Lambda function is ready for this scenario when its state is active and
its LastUpdateStatus is Successful.
     * Additionally, if a sha256 is provided, the function must have that as its
current code hash.
     * Any missing properties or failed requests will be reported as an Err.
     */
    async fn is_function_ready(
        &self,
        expected_code_sha256: Option<&str>,
    ) -> Result<bool, anyhow::Error> {
```

```
        match self.get_function().await {
            Ok(func) => {
                if let Some(config) = func.configuration() {
                    if let Some(state) = config.state() {
                        info!(?state, "Checking if function is active");
                        if !matches!(state, State::Active) {
                            return Ok(false);
                        }
                    }
                    match config.last_update_status() {
                        Some(last_update_status) => {
                            info!(?last_update_status, "Checking if function is
ready");
                            match last_update_status {
                                LastUpdateStatus::Successful => {
                                    // continue
                                }
                                LastUpdateStatus::Failed |
LastUpdateStatus::InProgress => {
                                    return Ok(false);
                                }
                                unknown => {
                                    warn!(
                                        status_variant = unknown.as_str(),
                                        "LastUpdateStatus unknown"
                                    );
                                    return Err(anyhow!(
                                        "Unknown LastUpdateStatus, fn config is
{config:?}"
                                    ));
                                }
                            }
                        }
                        None => {
                            warn!("Missing last update status");
                            return Ok(false);
                        }
                    };
                    if expected_code_sha256.is_none() {
                        return Ok(true);
                    }
                    if let Some(code_sha256) = config.code_sha256() {
                        return Ok(code_sha256 ==
expected_code_sha256.unwrap_or_default());
```

```
                    }
                }
            }
            Err(e) => {
                warn!(?e, "Could not get function while waiting");
            }
        }
        Ok(false)
    }

    /** Get the Lambda function with this Manager's name. */
    pub async fn get_function(&self) -> Result<GetFunctionOutput, anyhow::Error>
{
        info!("Getting lambda function");
        self.lambda_client
            .get_function()
            .function_name(self.lambda_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from)
    }

    /** List all Lambda functions in the current Region. */
    pub async fn list_functions(&self) -> Result<ListFunctionsOutput,
anyhow::Error> {
        info!("Listing lambda functions");
        self.lambda_client
            .list_functions()
            .send()
            .await
            .map_err(anyhow::Error::from)
    }

    /** Invoke the lambda function using calculator InvokeArgs. */
    pub async fn invoke(&self, args: InvokeArgs) -> Result<InvokeOutput,
anyhow::Error> {
        info!(?args, "Invoking {}", self.lambda_name);
        let payload = serde_json::to_string(&args)?;
        debug!(?payload, "Sending payload");
        self.lambda_client
            .invoke()
            .function_name(self.lambda_name.clone())
            .payload(Blob::new(payload))
            .send()
```

```
            .await
            .map_err(anyhow::Error::from)
    }

    /** Given a Path to a zip file, update the function's code and wait for the
update to finish. */
    pub async fn update_function_code(
        &self,
        zip_file: PathBuf,
        key: String,
    ) -> Result<UpdateFunctionCodeOutput, anyhow::Error> {
        let function_code = self.prepare_function(zip_file, Some(key)).await?;

        info!("Updating code for {}", self.lambda_name);
        let update = self
            .lambda_client
            .update_function_code()
            .function_name(self.lambda_name.clone())
            .s3_bucket(self.bucket.clone())
            .s3_key(function_code.s3_key().unwrap().to_string())
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        Ok(update)
    }

    /** Update the environment for a function. */
    pub async fn update_function_configuration(
        &self,
        environment: Environment,
    ) -> Result<UpdateFunctionConfigurationOutput, anyhow::Error> {
        info!(
            ?environment,
            "Updating environment for {}", self.lambda_name
        );
        let updated = self
            .lambda_client
            .update_function_configuration()
            .function_name(self.lambda_name.clone())
            .environment(environment)
            .send()
```

```
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        Ok(updated)
    }

    /** Delete a function and its role, and if possible or necessary, its
associated code object and bucket. */
    pub async fn delete_function(
        &self,
        location: Option<String>,
    ) -> (
        Result<DeleteFunctionOutput, anyhow::Error>,
        Result<DeleteRoleOutput, anyhow::Error>,
        Option<Result<DeleteObjectOutput, anyhow::Error>>,
    ) {
        info!("Deleting lambda function {}", self.lambda_name);
        let delete_function = self
            .lambda_client
            .delete_function()
            .function_name(self.lambda_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from);

        info!("Deleting iam role {}", self.role_name);
        let delete_role = self
            .iam_client
            .delete_role()
            .role_name(self.role_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from);

        let delete_object: Option<Result<DeleteObjectOutput, anyhow::Error>> =
            if let Some(location) = location {
                info!("Deleting object {location}");
                Some(
                    self.s3_client
                        .delete_object()
                        .bucket(self.bucket.clone())
                        .key(location)
```

```
                        .send()
                        .await
                        .map_err(anyhow::Error::from),
                )
            } else {
                info!(?location, "Skipping delete object");
                None
            };

        (delete_function, delete_role, delete_object)
    }

    pub async fn cleanup(
        &self,
        location: Option<String>,
    ) -> (
        (
            Result<DeleteFunctionOutput, anyhow::Error>,
            Result<DeleteRoleOutput, anyhow::Error>,
            Option<Result<DeleteObjectOutput, anyhow::Error>>,
        ),
        Option<Result<DeleteBucketOutput, anyhow::Error>>,
    ) {
        let delete_function = self.delete_function(location).await;

        let delete_bucket = if self.own_bucket {
            info!("Deleting bucket {}", self.bucket);
            if delete_function.2.is_none() ||
delete_function.2.as_ref().unwrap().is_ok() {
                Some(
                    self.s3_client
                        .delete_bucket()
                        .bucket(self.bucket.clone())
                        .send()
                        .await
                        .map_err(anyhow::Error::from),
                )
            } else {
                None
            }
        } else {
            info!("No bucket to clean up");
            None
        };
```

```
        (delete_function, delete_bucket)
    }
}

/**
 * Testing occurs primarily as an integration test running the `scenario` bin
 successfully.
 * Each action relies deeply on the internal workings and state of Amazon Simple
 Storage Service (Amazon S3), Lambda, and IAM working together.
 * It is therefore infeasible to mock the clients to test the individual actions.
 */
#[cfg(test)]
mod test {
    use super::{InvokeArgs, Operation};
    use serde_json::json;

    /** Make sure that the JSON output of serializing InvokeArgs is what's
 expected by the calculator. */
    #[test]
    fn test_serialize() {
        assert_eq!(json!(InvokeArgs::Increment(5)), 5);
        assert_eq!(
            json!(InvokeArgs::Arithmetic(Operation::Plus, 5, 7)).to_string(),
            r#"{"op":"plus","i":5,"j":7}"#.to_string(),
        );
    }
}
```

Un binaire pour exécuter le scénario de bout en bout, en utilisant des indicateurs de ligne de commande pour contrôler certains comportements. Ce fichier est src/bin/scenario .rs dans la caisse.

```
/*
## Service actions

Service actions wrap the SDK call, taking a client and any specific parameters
 necessary for the call.

* CreateFunction
* GetFunction
```

```
* ListFunctions
* Invoke
* UpdateFunctionCode
* UpdateFunctionConfiguration
* DeleteFunction

## Scenario
A scenario runs at a command prompt and prints output to the user on the result
 of each service action. A scenario can run in one of two ways: straight through,
 printing out progress as it goes, or as an interactive question/answer script.

## Getting started with functions

Use an SDK to manage AWS Lambda functions: create a function, invoke it, update
 its code, invoke it again, view its output and logs, and delete it.

This scenario uses two Lambda handlers:
_Note: Handlers don't use AWS SDK API calls._

The increment handler is straightforward:

1. It accepts a number, increments it, and returns the new value.
2. It performs simple logging of the result.

The arithmetic handler is more complex:
1. It accepts a set of actions ['plus', 'minus', 'times', 'divided-by'] and two
 numbers, and returns the result of the calculation.
2. It uses an environment variable to control log level (such as DEBUG, INFO,
 WARNING, ERROR).
It logs a few things at different levels, such as:
    * DEBUG: Full event data.
    * INFO: The calculation result.
    * WARN~ING~: When a divide by zero error occurs.
    * This will be the typical `RUST_LOG` variable.


The steps of the scenario are:

1. Create an AWS Identity and Access Management (IAM) role that meets the
 following requirements:
    * Has an assume_role policy that grants 'lambda.amazonaws.com' the
 'sts:AssumeRole' action.
    * Attaches the 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole' managed role.
```

```
    * _You must wait for ~10 seconds after the role is created before you can use
 it!_
2. Create a function (CreateFunction) for the increment handler by packaging it
 as a zip and doing one of the following:
    * Adding it with CreateFunction Code.ZipFile.
    * --or--
    * Uploading it to Amazon Simple Storage Service (Amazon S3) and adding it
 with CreateFunction Code.S3Bucket/S3Key.
    * _Note: Zipping the file does not have to be done in code._
    * If you have a waiter, use it to wait until the function is active.
 Otherwise, call GetFunction until State is Active.
3. Invoke the function with a number and print the result.
4. Update the function (UpdateFunctionCode) to the arithmetic handler by
 packaging it as a zip and doing one of the following:
    * Adding it with UpdateFunctionCode ZipFile.
    * --or--
    * Uploading it to Amazon S3 and adding it with UpdateFunctionCode S3Bucket/
S3Key.
5. Call GetFunction until Configuration.LastUpdateStatus is 'Successful' (or
 'Failed').
6. Update the environment variable by calling UpdateFunctionConfiguration and
 pass it a log level, such as:
    * Environment={'Variables': {'RUST_LOG': 'TRACE'}}
7. Invoke the function with an action from the list and a couple of values.
 Include LogType='Tail' to get logs in the result. Print the result of the
 calculation and the log.
8. [Optional] Invoke the function to provoke a divide-by-zero error and show the
 log result.
9. List all functions for the account, using pagination (ListFunctions).
10. Delete the function (DeleteFunction).
11. Delete the role.

Each step should use the function created in Service Actions to abstract calling
 the SDK.
 */

use aws_sdk_lambda::{operation::invoke::InvokeOutput, types::Environment};
use clap::Parser;
use std::{collections::HashMap, path::PathBuf};
use tracing::{debug, info, warn};
use tracing_subscriber::EnvFilter;

use lambda_code_examples::actions::{
    InvokeArgs::{Arithmetic, Increment},
```

```
    LambdaManager, Operation,
};

#[derive(Debug, Parser)]
pub struct Opt {
    /// The AWS Region.
    #[structopt(short, long)]
    pub region: Option<String>,

    // The bucket to use for the FunctionCode.
    #[structopt(short, long)]
    pub bucket: Option<String>,

    // The name of the Lambda function.
    #[structopt(short, long)]
    pub lambda_name: Option<String>,

    // The number to increment.
    #[structopt(short, long, default_value = "12")]
    pub inc: i32,

    // The left operand.
    #[structopt(long, default_value = "19")]
    pub num_a: i32,

    // The right operand.
    #[structopt(long, default_value = "23")]
    pub num_b: i32,

    // The arithmetic operation.
    #[structopt(short, long, default_value = "plus")]
    pub operation: Operation,

    #[structopt(long)]
    pub cleanup: Option<bool>,

    #[structopt(long)]
    pub no_cleanup: Option<bool>,
}

fn code_path(lambda: &str) -> PathBuf {
    PathBuf::from(format!("../target/lambda/{lambda}/bootstrap.zip"))
}
```

```
fn log_invoke_output(invoke: &InvokeOutput, message: &str) {
    if let Some(payload) = invoke.payload().cloned() {
        let payload = String::from_utf8(payload.into_inner());
        info!(?payload, message);
    } else {
        info!("Could not extract payload")
    }
    if let Some(logs) = invoke.log_result() {
        debug!(?logs, "Invoked function logs")
    } else {
        debug!("Invoked function had no logs")
    }
}

async fn main_block(
    opt: &Opt,
    manager: &LambdaManager,
    code_location: String,
) -> Result<(), anyhow::Error> {
    let invoke = manager.invoke(Increment(opt.inc)).await?;
    log_invoke_output(&invoke, "Invoked function configured as increment");

    let update_code = manager
        .update_function_code(code_path("arithmetic"), code_location.clone())
        .await?;

    let code_sha256 = update_code.code_sha256().unwrap_or("Unknown SHA");
    info!(?code_sha256, "Updated function code with arithmetic.zip");

    let arithmetic_args = Arithmetic(opt.operation, opt.num_a, opt.num_b);
    let invoke = manager.invoke(arithmetic_args).await?;
    log_invoke_output(&invoke, "Invoked function configured as arithmetic");

    let update = manager
        .update_function_configuration(
            Environment::builder()
                .set_variables(Some(HashMap::from([(
                    "RUST_LOG".to_string(),
                    "trace".to_string(),
                )])))
                .build(),
        )
        .await?;
    let updated_environment = update.environment();
```

```
    info!(?updated_environment, "Updated function configuration");

    let invoke = manager
        .invoke(Arithmetic(opt.operation, opt.num_a, opt.num_b))
        .await?;
    log_invoke_output(
        &invoke,
        "Invoked function configured as arithmetic with increased logging",
    );

    let invoke = manager
        .invoke(Arithmetic(Operation::DividedBy, opt.num_a, 0))
        .await?;
    log_invoke_output(
        &invoke,
        "Invoked function configured as arithmetic with divide by zero",
    );

    Ok::<(), anyhow::Error>(())
}

#[tokio::main]
async fn main() {
    tracing_subscriber::fmt()
        .without_time()
        .with_file(true)
        .with_line_number(true)
        .with_env_filter(EnvFilter::from_default_env())
        .init();

    let opt = Opt::parse();
    let manager = LambdaManager::load_from_env(opt.lambda_name.clone(),
 opt.bucket.clone()).await;

    let key = match manager.create_function(code_path("increment")).await {
        Ok(init) => {
            info!(?init, "Created function, initially with increment.zip");
            let run_block = main_block(&opt, &manager, init.clone()).await;
            info!(?run_block, "Finished running example, cleaning up");
            Some(init)
        }
        Err(err) => {
            warn!(?err, "Error happened when initializing function");
            None
```

```
        }
    };

    if Some(false) == opt.cleanup || Some(true) == opt.no_cleanup {
        info!("Skipping cleanup")
    } else {
        let delete = manager.cleanup(key).await;
        info!(?delete, "Deleted function & cleaned up resources");
    }
}
```

- Pour plus de détails sur l'API, consultez les rubriques suivantes dans la Référence des API du kit AWS SDK pour Rust.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

SAP ABAP

Kit SDK pour SAP ABAP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
   TRY.
       "Create an AWS Identity and Access Management (IAM) role that grants AWS
Lambda permission to write to logs."
       DATA(lv_policy_document) = `{` &&
           `"Version":"2012-10-17",       ` &&
```

```
                  `"Statement": [` &&
                    `{` &&
                      `"Effect": "Allow",` &&
                      `"Action": [` &&
                        `"sts:AssumeRole"` &&
                      `],` &&
                      `"Principal": {` &&
                        `"Service": [` &&
                          `"lambda.amazonaws.com"` &&
                        `]` &&
                      `}` &&
                    `}` &&
                  `]` &&
                `}`.
        TRY.
            DATA(lo_create_role_output) = lo_iam->createrole(
                    iv_rolename = iv_role_name
                    iv_assumerolepolicydocument = lv_policy_document
                    iv_description = 'Grant lambda permission to write to
 logs' ).
            DATA(lv_role_arn) = lo_create_role_output->get_role( )->get_arn( ).
            MESSAGE 'IAM role created.' TYPE 'I'.
            WAIT UP TO 10 SECONDS.            " Make sure that the IAM role is
 ready for use. "
          CATCH /aws1/cx_iamentityalrdyexex.
            DATA(lo_role) = lo_iam->getrole( iv_rolename = iv_role_name ).
            lv_role_arn = lo_role->get_role( )->get_arn( ).
          CATCH /aws1/cx_iaminvalidinputex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_iammalformedplydocex.
            MESSAGE 'Policy document in the request is malformed.' TYPE 'E'.
        ENDTRY.

        TRY.
            lo_iam->attachrolepolicy(
                iv_rolename  = iv_role_name
                iv_policyarn = 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole' ).
            MESSAGE 'Attached policy to the IAM role.' TYPE 'I'.
          CATCH /aws1/cx_iaminvalidinputex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_iamnosuchentityex.
            MESSAGE 'The requested resource entity does not exist.' TYPE 'E'.
          CATCH /aws1/cx_iamplynotattachableex.
```

```
            MESSAGE 'Service role policies can only be attached to the service-
linked role for their service.' TYPE 'E'.
          CATCH /aws1/cx_iamunmodableentityex.
            MESSAGE 'Service that depends on the service-linked role is not
 modifiable.' TYPE 'E'.
        ENDTRY.

        " Create a Lambda function and upload handler code. "
        " Lambda function performs 'increment' action on a number. "
        TRY.
            lo_lmd->createfunction(
                 iv_functionname = iv_function_name
                 iv_runtime = `python3.9`
                 iv_role = lv_role_arn
                 iv_handler = iv_handler
                 io_code = io_initial_zip_file
                 iv_description = 'AWS Lambda code example' ).
            MESSAGE 'Lambda function created.' TYPE 'I'.
          CATCH /aws1/cx_lmdcodestorageexcdex.
            MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
        ENDTRY.

        " Verify the function is in Active state "
        WHILE lo_lmd->getfunction( iv_functionname = iv_function_name )-
>get_configuration( )->ask_state( ) <> 'Active'.
          IF sy-index = 10.
            EXIT.               " Maximum 10 seconds. "
          ENDIF.
          WAIT UP TO 1 SECONDS.
        ENDWHILE.

        "Invoke the function with a single parameter and get results."
        TRY.
            DATA(lv_json) = /aws1/cl_rt_util=>string_to_xstring(
              `{`  &&
                `"action": "increment",`  &&
                `"number": 10` &&
              `}` ).
            DATA(lo_initial_invoke_output) = lo_lmd->invoke(
                       iv_functionname = iv_function_name
```

```
                        iv_payload = lv_json ).
            ov_initial_invoke_payload = lo_initial_invoke_output->get_payload( ).
           " ov_initial_invoke_payload is returned for testing purposes. "
            DATA(lo_writer_json) = cl_sxml_string_writer=>create( type =
 if_sxml=>co_xt_json ).
            CALL TRANSFORMATION id SOURCE XML ov_initial_invoke_payload RESULT
 XML lo_writer_json.
            DATA(lv_result) = cl_abap_codepage=>convert_from( lo_writer_json-
>get_output( ) ).
            MESSAGE 'Lambda function invoked.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvrequestcontex.
            MESSAGE 'Unable to parse request body as JSON.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
          CATCH /aws1/cx_lmdunsuppedmediatyp00.
            MESSAGE 'Invoke request body does not have JSON as its content type.'
 TYPE 'E'.
        ENDTRY.

        " Update the function code and configure its Lambda environment with an
 environment variable. "
        " Lambda function is updated to perform 'decrement' action also. "
        TRY.
            lo_lmd->updatefunctioncode(
                  iv_functionname = iv_function_name
                  iv_zipfile = io_updated_zip_file ).
            WAIT UP TO 10 SECONDS.            " Make sure that the update is
 completed. "
            MESSAGE 'Lambda function code updated.' TYPE 'I'.
          CATCH /aws1/cx_lmdcodestorageexcdex.
            MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
        ENDTRY.

        TRY.
            DATA lt_variables TYPE /aws1/
cl_lmdenvironmentvaria00=>tt_environmentvariables.
            DATA ls_variable LIKE LINE OF lt_variables.
            ls_variable-key = 'LOG_LEVEL'.
```

```
            ls_variable-value = NEW /aws1/cl_lmdenvironmentvaria00( iv_value =
 'info' ).
            INSERT ls_variable INTO TABLE lt_variables.

            lo_lmd->updatefunctionconfiguration(
                  iv_functionname = iv_function_name
                  io_environment = NEW /aws1/cl_lmdenvironment( it_variables =
 lt_variables ) ).
            WAIT UP TO 10 SECONDS.            " Make sure that the update is
 completed. "
            MESSAGE 'Lambda function configuration/settings updated.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourceconflictex.
            MESSAGE 'Resource already exists or another operation is in
 progress.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
        ENDTRY.

        "Invoke the function with new parameters and get results. Display the
 execution log that's returned from the invocation."
        TRY.
            lv_json = /aws1/cl_rt_util=>string_to_xstring(
              `{`  &&
                `"action": "decrement",`  &&
                `"number": 10` &&
              `}` ).
            DATA(lo_updated_invoke_output) = lo_lmd->invoke(
                       iv_functionname = iv_function_name
                       iv_payload = lv_json ).
            ov_updated_invoke_payload = lo_updated_invoke_output->get_payload( ).
           " ov_updated_invoke_payload is returned for testing purposes. "
            lo_writer_json = cl_sxml_string_writer=>create( type =
 if_sxml=>co_xt_json ).
            CALL TRANSFORMATION id SOURCE XML ov_updated_invoke_payload RESULT
 XML lo_writer_json.
            lv_result = cl_abap_codepage=>convert_from( lo_writer_json-
>get_output( ) ).
            MESSAGE 'Lambda function invoked.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvrequestcontex.
            MESSAGE 'Unable to parse request body as JSON.' TYPE 'E'.
```

```
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
          CATCH /aws1/cx_lmdunsuppedmediatyp00.
            MESSAGE 'Invoke request body does not have JSON as its content type.'
 TYPE 'E'.
        ENDTRY.

        " List the functions for your account. "
        TRY.
            DATA(lo_list_output) = lo_lmd->listfunctions( ).
            DATA(lt_functions) = lo_list_output->get_functions( ).
            MESSAGE 'Retrieved list of Lambda functions.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
        ENDTRY.

        " Delete the Lambda function. "
        TRY.
            lo_lmd->deletefunction( iv_functionname = iv_function_name ).
            MESSAGE 'Lambda function deleted.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'W'.
        ENDTRY.

        " Detach role policy. "
        TRY.
            lo_iam->detachrolepolicy(
                iv_rolename  = iv_role_name
                iv_policyarn = 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole' ).
            MESSAGE 'Detached policy from the IAM role.' TYPE 'I'.
          CATCH /aws1/cx_iaminvalidinputex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_iamnosuchentityex.
            MESSAGE 'The requested resource entity does not exist.' TYPE 'W'.
          CATCH /aws1/cx_iamplynotattachableex.
            MESSAGE 'Service role policies can only be attached to the service-
linked role for their service.' TYPE 'E'.
          CATCH /aws1/cx_iamunmodableentityex.
            MESSAGE 'Service that depends on the service-linked role is not
 modifiable.' TYPE 'E'.
        ENDTRY.
```

```
        " Delete the IAM role. "
        TRY.
            lo_iam->deleterole( iv_rolename = iv_role_name ).
            MESSAGE 'IAM role deleted.' TYPE 'I'.
          CATCH /aws1/cx_iamnosuchentityex.
            MESSAGE 'The requested resource entity does not exist.' TYPE 'W'.
          CATCH /aws1/cx_iamunmodableentityex.
            MESSAGE 'Service that depends on the service-linked role is not
modifiable.' TYPE 'E'.
        ENDTRY.

      CATCH /aws1/cx_rt_service_generic INTO lo_exception.
        DATA(lv_error) = lo_exception->get_longtext( ).
        MESSAGE lv_error TYPE 'E'.
    ENDTRY.
```

- Pour plus de détails sur l'API, consultez les rubriques suivantes dans la Référence des API du kit AWS SDK pour SAP ABAP.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Swift

Kit SDK pour Swift

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

Définissez la première fonction Lambda, qui incrémente simplement la valeur spécifiée.

```
// swift-tools-version: 5.9
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
//
// The swift-tools-version declares the minimum version of Swift required to
// build this package.

import PackageDescription

let package = Package(
    name: "increment",
    // Let Xcode know the minimum Apple platforms supported.
    platforms: [
        .macOS(.v13)
    ],
    dependencies: [
        // Dependencies declare other packages that this package depends on.
        .package(
            url: "https://github.com/swift-server/swift-aws-lambda-runtime.git",
            branch: "main"),
    ],
    targets: [
        // Targets are the basic building blocks of a package, defining a module
 or a test suite.
        // Targets can depend on other targets in this package and products
        // from dependencies.
        .executableTarget(
            name: "increment",
            dependencies: [
                .product(name: "AWSLambdaRuntime", package: "swift-aws-lambda-
runtime"),
            ],
            path: "Sources"
        )
    ]
)

import Foundation
import AWSLambdaRuntime

/// Represents the contents of the requests being received from the client.
/// This structure must be `Decodable` to indicate that its initializer
/// converts an external representation into this type.
```

```
struct Request: Decodable, Sendable {
    /// The action to perform.
    let action: String
    /// The number to act upon.
    let number: Int
}

/// The contents of the response sent back to the client. This must be
/// `Encodable`.
struct Response: Encodable, Sendable {
    /// The resulting value after performing the action.
    let answer: Int?
}


/// The Lambda function body.
///
/// - Parameters:
///   - event: The `Request` describing the request made by the
///     client.
///   - context: A `LambdaContext` describing the context in
///     which the lambda function is running.
///
/// - Returns: A `Response` object that will be encoded to JSON and sent
///   to the client by the Lambda runtime.
let incrementLambdaRuntime = LambdaRuntime {
        (event: Request, context: LambdaContext) -> Response in
    let action = event.action
    var answer: Int?

    if action != "increment" {
        context.logger.error("Unrecognized operation: \"\(action)\". The only
 supported action is \"increment\".")
    } else {
        answer = event.number + 1
        context.logger.info("The calculated answer is \(answer!).")
    }

    let response = Response(answer: answer)
    return response
}

// Run the Lambda runtime code.
```

```
try await incrementLambdaRuntime.run()
```

Définissez la deuxième fonction Lambda, qui effectue une opération arithmétique sur deux nombres.

```
// swift-tools-version: 5.9
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
//
// The swift-tools-version declares the minimum version of Swift required to
// build this package.

import PackageDescription

let package = Package(
    name: "calculator",
    // Let Xcode know the minimum Apple platforms supported.
    platforms: [
        .macOS(.v13)
    ],
    dependencies: [
        // Dependencies declare other packages that this package depends on.
        .package(
            url: "https://github.com/swift-server/swift-aws-lambda-runtime.git",
            branch: "main"),
    ],
    targets: [
        // Targets are the basic building blocks of a package, defining a module
 or a test suite.
        // Targets can depend on other targets in this package and products
        // from dependencies.
        .executableTarget(
            name: "calculator",
            dependencies: [
                .product(name: "AWSLambdaRuntime", package: "swift-aws-lambda-
runtime"),
            ],
            path: "Sources"
        )
    ]
)
```

```
import Foundation
import AWSLambdaRuntime

/// Represents the contents of the requests being received from the client.
/// This structure must be `Decodable` to indicate that its initializer
/// converts an external representation into this type.
struct Request: Decodable, Sendable {
    /// The action to perform.
    let action: String
    /// The first number to act upon.
    let x: Int
    /// The second number to act upon.
    let y: Int
}

/// A dictionary mapping operation names to closures that perform that
/// operation and return the result.
let actions = [
    "plus": { (x: Int, y: Int) -> Int in
        return x + y
    },
    "minus": { (x: Int, y: Int) -> Int in
        return x - y
    },
    "times": { (x: Int, y: Int) -> Int in
        return x * y
    },
    "divided-by": { (x: Int, y: Int) -> Int in
        return x / y
    }
]

/// The contents of the response sent back to the client. This must be
/// `Encodable`.
struct Response: Encodable, Sendable {
    /// The resulting value after performing the action.
    let answer: Int?
}


/// The Lambda function's entry point. Called by the Lambda runtime.
///
/// - Parameters:
///   - event: The `Request` describing the request made by the
```

```
///     client.
///   - context: A `LambdaContext` describing the context in
///     which the lambda function is running.
///
/// - Returns: A `Response` object that will be encoded to JSON and sent
///   to the client by the Lambda runtime.
let calculatorLambdaRuntime = LambdaRuntime {
        (_ event: Request, context: LambdaContext) -> Response in
    let action = event.action
    var answer: Int?
    var actionFunc: ((Int, Int) -> Int)?

    // Get the closure to run to perform the calculation.

    actionFunc = await actions[action]

    guard let actionFunc else {
        context.logger.error("Unrecognized operation '\(action)\'")
        return Response(answer: nil)
    }

    // Perform the calculation and return the answer.

    answer = actionFunc(event.x, event.y)

    guard let answer else {
        context.logger.error("Error computing \(event.x) \(action) \(event.y)")
    }
    context.logger.info("\(event.x) \(action) \(event.y) = \(answer)")

    return Response(answer: answer)
}

try await calculatorLambdaRuntime.run()
```

Définissez le programme principal qui invoquera les deux fonctions Lambda.

```
// swift-tools-version: 5.9
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
//
// The swift-tools-version declares the minimum version of Swift required to
```

```
// build this package.

import PackageDescription

let package = Package(
    name: "lambda-basics",
    // Let Xcode know the minimum Apple platforms supported.
    platforms: [
        .macOS(.v13)
    ],
    dependencies: [
        // Dependencies declare other packages that this package depends on.
        .package(
            url: "https://github.com/awslabs/aws-sdk-swift",
            from: "1.0.0"),
        .package(
            url: "https://github.com/apple/swift-argument-parser.git",
            branch: "main"
        )
    ],
    targets: [
        // Targets are the basic building blocks of a package, defining a module
 or a test suite.
        // Targets can depend on other targets in this package and products
        // from dependencies.
        .executableTarget(
            name: "lambda-basics",
            dependencies: [
                .product(name: "AWSLambda", package: "aws-sdk-swift"),
                .product(name: "AWSIAM", package: "aws-sdk-swift"),
                .product(name: "ArgumentParser", package: "swift-argument-
parser")
            ],
            path: "Sources"
        )
    ]
)

//
/// An example demonstrating a variety of important AWS Lambda functions.

import ArgumentParser
import AWSIAM
import SmithyWaitersAPI
```

```
import AWSClientRuntime
import AWSLambda
import Foundation

/// Represents the contents of the requests being received from the client.
/// This structure must be `Decodable` to indicate that its initializer
/// converts an external representation into this type.
struct IncrementRequest: Encodable, Decodable, Sendable {
    /// The action to perform.
    let action: String
    /// The number to act upon.
    let number: Int
}

struct Response: Encodable, Decodable, Sendable {
    /// The resulting value after performing the action.
    let answer: Int?
}

struct CalculatorRequest: Encodable, Decodable, Sendable {
    /// The action to perform.
    let action: String
    /// The first number to act upon.
    let x: Int
    /// The second number to act upon.
    let y: Int
}

let exampleName = "SwiftLambdaRoleExample"
let basicsFunctionName = "lambda-basics-function"

/// The ARN of the standard IAM policy for execution of Lambda functions.
let policyARN = "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole"

struct ExampleCommand: ParsableCommand {
    // -MARK: Command arguments
    @Option(help: "Name of the IAM Role to use for the Lambda functions")
    var role = exampleName
    @Option(help: "Zip archive containing the 'increment' lambda function")
    var incpath: String
    @Option(help: "Zip archive containing the 'calculator' lambda function")
    var calcpath: String
    @Option(help: "Name of the Amazon S3 Region to use (default: us-east-1)")
```

```
    var region = "us-east-1"

    static var configuration = CommandConfiguration(
        commandName: "lambda-basics",
        abstract: """
        This example demonstrates several common operations using AWS Lambda.
        """,
        discussion: """
        """
    )

    /// Returns the specified IAM role object.
    ///
    /// - Parameters:
    ///   - iamClient: `IAMClient` to use when looking for the role.
    ///   - roleName: The name of the role to check.
    ///
    /// - Returns: The `IAMClientTypes.Role` representing the specified role.
    func getRole(iamClient: IAMClient, roleName: String) async throws
                -> IAMClientTypes.Role {
        do {
            let roleOutput = try await iamClient.getRole(
                input: GetRoleInput(
                    roleName: roleName
                )
            )

            guard let role = roleOutput.role else {
                throw ExampleError.roleNotFound
            }
            return role
        } catch {
            throw ExampleError.roleNotFound
        }
    }

    /// Create the AWS IAM role that will be used to access AWS Lambda.
    ///
    /// - Parameters:
    ///   - iamClient: The AWS `IAMClient` to use.
    ///   - roleName: The name of the AWS IAM role to use for Lambda.
    ///
    /// - Throws: `ExampleError.roleCreateError`
    ///
```

```
    /// - Returns: The `IAMClientTypes.Role` struct that describes the new role.
    func createRoleForLambda(iamClient: IAMClient, roleName: String) async throws
-> IAMClientTypes.Role {
        let output = try await iamClient.createRole(
            input: CreateRoleInput(
                assumeRolePolicyDocument:
                """
                {
                    "Version":"2012-10-17",
                    "Statement": [
                        {
                            "Effect": "Allow",
                            "Principal": {"Service": "lambda.amazonaws.com"},
                            "Action": "sts:AssumeRole"
                        }
                    ]
                }
                """,
                roleName: roleName
            )
        )

        guard let role = output.role else {
            throw ExampleError.roleCreateError
        }

        // Wait for the role to be ready for use.

        _ = try await iamClient.waitUntilRoleExists(
            options: WaiterOptions(
                maxWaitTime: 20,
                minDelay: 0.5,
                maxDelay: 2
            ),
            input: GetRoleInput(roleName: roleName)
        )

        return role
    }

    /// Detect whether or not the AWS Lambda function with the specified name
    /// exists, by requesting its function information.
    ///
    /// - Parameters:
```

```
    ///   - lambdaClient: The `LambdaClient` to use.
    ///   - name: The name of the AWS Lambda function to find.
    ///
    /// - Returns: `true` if the Lambda function exists. Otherwise `false`.
    func doesLambdaFunctionExist(lambdaClient: LambdaClient, name: String) async
-> Bool {
        do {
            _ = try await lambdaClient.getFunction(
                input: GetFunctionInput(functionName: name)
            )
        } catch {
            return false
        }

        return true
    }

    /// Create the specified AWS Lambda function.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `LambdaClient` to use.
    ///   - functionName: The name of the AWS Lambda function to create.
    ///   - roleArn: The ARN of the role to apply to the function.
    ///   - path: The path of the Zip archive containing the function.
    ///
    /// - Returns: `true` if the AWS Lambda was successfully created; `false`
    ///   if it wasn't.
    func createFunction(lambdaClient: LambdaClient, functionName: String,
                                roleArn: String?, path: String) async throws ->
Bool {
        do {
            // Read the Zip archive containing the AWS Lambda function.

            let zipUrl = URL(fileURLWithPath: path)
            let zipData = try Data(contentsOf: zipUrl)

            // Create the AWS Lambda function that runs the specified code,
            // using the name given on the command line. The Lambda function
            // will run using the Amazon Linux 2 runtime.

            _ = try await lambdaClient.createFunction(
                input: CreateFunctionInput(
                    code: LambdaClientTypes.FunctionCode(zipFile: zipData),
                    functionName: functionName,
```

```
                    handler: "handle",
                    role: roleArn,
                    runtime: .providedal2
                )
            )
        } catch {
            print("*** Error creating Lambda function:")
            dump(error)
            return false
        }

        // Wait for a while to be sure the function is done being created.

        let output = try await lambdaClient.waitUntilFunctionActiveV2(
            options: WaiterOptions(
                maxWaitTime: 20,
                minDelay: 0.5,
                maxDelay: 2
            ),
            input: GetFunctionInput(functionName: functionName)
        )

        switch output.result {
            case .success:
                return true
            case .failure:
                return false
        }
    }

    /// Update the AWS Lambda function with new code to run when the function
    /// is invoked.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `LambdaClient` to use.
    ///   - functionName: The name of the AWS Lambda function to update.
    ///   - path: The pathname of the Zip file containing the packaged Lambda
    ///     function.
    /// - Throws: `ExampleError.zipFileReadError`
    /// - Returns: `true` if the function's code is updated successfully.
    ///   Otherwise, returns `false`.
    func updateFunctionCode(lambdaClient: LambdaClient, functionName: String,
                            path: String) async throws -> Bool {
        let zipUrl = URL(fileURLWithPath: path)
```

```
        let zipData: Data

        // Read the function's Zip file.

        do {
            zipData = try Data(contentsOf: zipUrl)
        } catch {
            throw ExampleError.zipFileReadError
        }

        // Update the function's code and wait for the updated version to be
        // ready for use.

        do {
            _ = try await lambdaClient.updateFunctionCode(
                input: UpdateFunctionCodeInput(
                    functionName: functionName,
                    zipFile: zipData
                )
            )
        } catch {
            return false
        }

        let output = try await lambdaClient.waitUntilFunctionUpdatedV2(
            options: WaiterOptions(
                maxWaitTime: 20,
                minDelay: 0.5,
                maxDelay: 2
            ),
            input: GetFunctionInput(
                functionName: functionName
            )
        )

        switch output.result {
            case .success:
                return true
            case .failure:
                return false
        }
    }

    /// Tell the server-side component to log debug output by setting its
```

```
    /// environment's `LOG_LEVEL` to `DEBUG`.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `LambdaClient` to use.
    ///   - functionName: The name of the AWS Lambda function to enable debug
    ///     logging for.
    ///
    /// - Throws: `ExampleError.environmentResponseMissingError`,
    ///   `ExampleError.updateFunctionConfigurationError`,
    ///   `ExampleError.environmentVariablesMissingError`,
    ///   `ExampleError.logLevelIncorrectError`,
    ///   `ExampleError.updateFunctionConfigurationError`
    func enableDebugLogging(lambdaClient: LambdaClient, functionName: String)
async throws {
        let envVariables = [
            "LOG_LEVEL": "DEBUG"
        ]
        let environment = LambdaClientTypes.Environment(variables: envVariables)

        do {
            let output = try await lambdaClient.updateFunctionConfiguration(
                input: UpdateFunctionConfigurationInput(
                    environment: environment,
                    functionName: functionName
                )
            )

            guard let response = output.environment else {
                throw ExampleError.environmentResponseMissingError
            }

            if response.error != nil {
                throw ExampleError.updateFunctionConfigurationError
            }

            guard let retVariables = response.variables else {
                throw ExampleError.environmentVariablesMissingError
            }

            for envVar in retVariables {
                if envVar.key == "LOG_LEVEL" && envVar.value != "DEBUG" {
                    print("*** Log level is not set to DEBUG!")
                    throw ExampleError.logLevelIncorrectError
                }
```

```
            }
        } catch {
            throw ExampleError.updateFunctionConfigurationError
        }
    }

    /// Returns an array containing the names of all AWS Lambda functions
    /// available to the user.
    ///
    /// - Parameter lambdaClient: The `IAMClient` to use.
    ///
    /// - Throws: `ExampleError.listFunctionsError`
    ///
    /// - Returns: An array of lambda function name strings.
    func getFunctionNames(lambdaClient: LambdaClient) async throws -> [String] {
        let pages = lambdaClient.listFunctionsPaginated(
            input: ListFunctionsInput()
        )

        var functionNames: [String] = []

        for try await page in pages {
            guard let functions = page.functions else {
                throw ExampleError.listFunctionsError
            }

            for function in functions {
                functionNames.append(function.functionName ?? "<unknown>")
            }
        }

        return functionNames
    }

    /// Invoke the Lambda function to increment a value.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `IAMClient` to use.
    ///   - number: The number to increment.
    ///
    /// - Throws: `ExampleError.noAnswerReceived`, `ExampleError.invokeError`
    ///
    /// - Returns: An integer number containing the incremented value.
```

```
    func invokeIncrement(lambdaClient: LambdaClient, number: Int) async throws ->
Int {
        do {
            let incRequest = IncrementRequest(action: "increment", number:
number)
            let incData = try! JSONEncoder().encode(incRequest)

            // Invoke the lambda function.

            let invokeOutput = try await lambdaClient.invoke(
                input: InvokeInput(
                    functionName: "lambda-basics-function",
                    payload: incData
                )
            )

            let response = try! JSONDecoder().decode(Response.self,
from:invokeOutput.payload!)

            guard let answer = response.answer else {
                throw ExampleError.noAnswerReceived
            }
            return answer

        } catch {
            throw ExampleError.invokeError
        }
    }

    /// Invoke the calculator Lambda function.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `IAMClient` to use.
    ///   - action: Which arithmetic operation to perform: "plus", "minus",
    ///     "times", or "divided-by".
    ///   - x: The first number to use in the computation.
    ///   - y: The second number to use in the computation.
    ///
    /// - Throws: `ExampleError.noAnswerReceived`, `ExampleError.invokeError`
    ///
    /// - Returns: The computed answer as an `Int`.
    func invokeCalculator(lambdaClient: LambdaClient, action: String, x: Int, y:
Int) async throws -> Int {
        do {
```

```
            let calcRequest = CalculatorRequest(action: action, x: x, y: y)
            let calcData = try! JSONEncoder().encode(calcRequest)

            // Invoke the lambda function.

            let invokeOutput = try await lambdaClient.invoke(
                input: InvokeInput(
                    functionName: "lambda-basics-function",
                    payload: calcData
                )
            )

            let response = try! JSONDecoder().decode(Response.self,
from:invokeOutput.payload!)

            guard let answer = response.answer else {
                throw ExampleError.noAnswerReceived
            }
            return answer

        } catch {
            throw ExampleError.invokeError
        }

    }

    /// Perform the example's tasks.
    func basics() async throws {
        let iamClient = try await IAMClient(
            config: IAMClient.IAMClientConfiguration(region: region)
        )

        let lambdaClient = try await LambdaClient(
            config: LambdaClient.LambdaClientConfiguration(region: region)
        )

        /// The IAM role to use for the example.
        var iamRole: IAMClientTypes.Role

        // Look for the specified role. If it already exists, use it. If not,
        // create it and attach the desired policy to it.

        do {
            iamRole = try await getRole(iamClient: iamClient, roleName: role)
```

```
        } catch ExampleError.roleNotFound {
            // The role wasn't found, so create it and attach the needed
            // policy.

            iamRole = try await createRoleForLambda(iamClient: iamClient,
roleName: role)

            do {
                _ = try await iamClient.attachRolePolicy(
                    input: AttachRolePolicyInput(policyArn: policyARN, roleName:
role)
                )
            } catch {
                throw ExampleError.policyError
            }
        }

        // Give the policy time to attach to the role.

        sleep(5)

        // Look to see if the function already exists. If it does, throw an
        // error.

        if await doesLambdaFunctionExist(lambdaClient: lambdaClient, name:
basicsFunctionName) {
            throw ExampleError.functionAlreadyExists
        }

        // Create, then invoke, the "increment" version of the calculator
        // function.

        print("Creating the increment Lambda function...")
        if try await createFunction(lambdaClient: lambdaClient, functionName:
basicsFunctionName,
                                    roleArn: iamRole.arn, path: incpath) {
            print("Running increment function calls...")
            for number in 0...4 {
                do {
                    let answer = try await invokeIncrement(lambdaClient:
lambdaClient, number: number)
                    print("Increment \(number) = \(answer)")
                } catch {
```

```
                    print("Error incrementing \(number): ",
error.localizedDescription)
                }
            }
        } else {
            print("*** Failed to create the increment function.")
        }

        // Enable debug logging.

        print("\nEnabling debug logging...")
        try await enableDebugLogging(lambdaClient: lambdaClient, functionName:
basicsFunctionName)

        // Change it to a basic arithmetic calculator. Then invoke it a few
        // times.

        print("\nReplacing the Lambda function with a calculator...")

        if try await updateFunctionCode(lambdaClient: lambdaClient, functionName:
basicsFunctionName,
                                        path: calcpath) {
            print("Running calculator function calls...")
            for x in [6, 10] {
                for y in [2, 4] {
                    for action in ["plus", "minus", "times", "divided-by"] {
                        do {
                            let answer = try await invokeCalculator(lambdaClient:
lambdaClient, action: action, x: x, y: y)
                            print("\(x) \(action) \(y) = \(answer)")
                        } catch {
                            print("Error calculating \(x) \(action) \(y): ",
error.localizedDescription)
                        }
                    }
                }
            }
        }

        // List all lambda functions.

        let functionNames = try await getFunctionNames(lambdaClient:
lambdaClient)
```

```
        if functionNames.count > 0 {
            print("\nAWS Lambda functions available on your account:")
            for name in functionNames {
                print("  \(name)")
            }
        }

        // Delete the lambda function.

        print("Deleting lambda function...")

        do {
            _ = try await lambdaClient.deleteFunction(
                input: DeleteFunctionInput(
                    functionName: "lambda-basics-function"
                )
            )
        } catch {
            print("Error: Unable to delete the function.")
        }

        // Detach the role from the policy, then delete the role.

        print("Deleting the AWS IAM role...")

        do {
            _ = try await iamClient.detachRolePolicy(
                input: DetachRolePolicyInput(
                    policyArn: policyARN,
                    roleName: role
                )
            )
            _ = try await iamClient.deleteRole(
                input: DeleteRoleInput(
                    roleName: role
                )
            )
        } catch {
            throw ExampleError.deleteRoleError
        }
    }
}

// -MARK: - Entry point
```

```
/// The program's asynchronous entry point.
@main
struct Main {
    static func main() async {
        let args = Array(CommandLine.arguments.dropFirst())

        do {
            let command = try ExampleCommand.parse(args)
            try await command.basics()
        } catch {
            ExampleCommand.exit(withError: error)
        }
    }
}


/// Errors thrown by the example's functions.
enum ExampleError: Error {
    /// An AWS Lambda function with the specified name already exists.
    case functionAlreadyExists
    /// The specified role doesn't exist.
    case roleNotFound
    /// Unable to create the role.
    case roleCreateError
    /// Unable to delete the role.
    case deleteRoleError
    /// Unable to attach a policy to the role.
    case policyError
    /// Unable to get the executable directory.
    case executableNotFound
    /// An error occurred creating a lambda function.
    case createLambdaError
    /// An error occurred invoking the lambda function.
    case invokeError
    /// No answer received from the invocation.
    case noAnswerReceived
    /// Unable to list the AWS Lambda functions.
    case listFunctionsError
    /// Unable to update the AWS Lambda function.
    case updateFunctionError
    /// Unable to update the function configuration.
    case updateFunctionConfigurationError
    /// The environment response is missing after an
```

```
    /// UpdateEnvironmentConfiguration attempt.
    case environmentResponseMissingError
    /// The environment variables are missing from the EnvironmentResponse and
    /// no errors occurred.
    case environmentVariablesMissingError
    /// The log level is incorrect after attempting to set it.
    case logLevelIncorrectError
    /// Unable to load the AWS Lambda function's Zip file.
    case zipFileReadError

    var errorDescription: String? {
        switch self {
        case .functionAlreadyExists:
            return "An AWS Lambda function with that name already exists."
        case .roleNotFound:
            return "The specified role doesn't exist."
        case .deleteRoleError:
            return "Unable to delete the AWS IAM role."
        case .roleCreateError:
            return "Unable to create the specified role."
        case .policyError:
            return "An error occurred attaching the policy to the role."
        case .executableNotFound:
            return "Unable to find the executable program directory."
        case .createLambdaError:
            return "An error occurred creating a lambda function."
        case .invokeError:
            return "An error occurred invoking a lambda function."
        case .noAnswerReceived:
            return "No answer received from the lambda function."
        case .listFunctionsError:
            return "Unable to list the AWS Lambda functions."
        case .updateFunctionError:
            return "Unable to update the AWS lambda function."
        case .updateFunctionConfigurationError:
            return "Unable to update the AWS lambda function configuration."
        case .environmentResponseMissingError:
            return "The environment is missing from the response after updating
the function configuration."
        case .environmentVariablesMissingError:
            return "While no error occurred, no environment variables were
returned following function configuration."
        case .logLevelIncorrectError:
```

```
            return "The log level is incorrect after attempting to set it to
 DEBUG."
        case .zipFileReadError:
            return "Unable to read the AWS Lambda function."
        }
    }
}
```

- Pour plus de détails sur l'API, consultez les rubriques suivantes dans la Référence des API du kit AWS SDK pour Swift.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Actions pour Lambda utilisant AWS SDKs

Les exemples de code suivants montrent comment effectuer des actions Lambda individuelles avec. AWS SDKs Chaque exemple inclut un lien vers GitHub, où vous pouvez trouver des instructions pour configurer et exécuter le code.

Ces extraits appellent l'API Lambda et sont des extraits de code de programmes plus volumineux qui doivent être exécutés en contexte. Vous pouvez voir les actions dans leur contexte dans Scénarios d'utilisation de Lambda AWS SDKs .

Les exemples suivants incluent uniquement les actions les plus couramment utilisées. Pour obtenir la liste complète, consultez la Référence des API AWS Lambda.

Exemples

- Utilisation de CreateAlias avec une CLI

- Utilisation CreateFunction avec un AWS SDK ou une CLI
- Utilisation de DeleteAlias avec une CLI
- Utilisation DeleteFunction avec un AWS SDK ou une CLI
- Utilisation de DeleteFunctionConcurrency avec une CLI
- Utilisation de DeleteProvisionedConcurrencyConfig avec une CLI
- Utilisation de GetAccountSettings avec une CLI
- Utilisation de GetAlias avec une CLI
- Utilisation GetFunction avec un AWS SDK ou une CLI
- Utilisation de GetFunctionConcurrency avec une CLI
- Utilisation de GetFunctionConfiguration avec une CLI
- Utilisation de GetPolicy avec une CLI
- Utilisation de GetProvisionedConcurrencyConfig avec une CLI
- Utilisation Invoke avec un AWS SDK ou une CLI
- Utilisation ListFunctions avec un AWS SDK ou une CLI
- Utilisation de ListProvisionedConcurrencyConfigs avec une CLI
- Utilisation de ListTags avec une CLI
- Utilisation de ListVersionsByFunction avec une CLI
- Utilisation de PublishVersion avec une CLI
- Utilisation de PutFunctionConcurrency avec une CLI
- Utilisation de PutProvisionedConcurrencyConfig avec une CLI
- Utilisation de RemovePermission avec une CLI
- Utilisation de TagResource avec une CLI
- Utilisation de UntagResource avec une CLI
- Utilisation de UpdateAlias avec une CLI
- Utilisation UpdateFunctionCode avec un AWS SDK ou une CLI
- Utilisation UpdateFunctionConfiguration avec un AWS SDK ou une CLI

Utilisation de **`CreateAlias`** avec une CLI

Les exemples de code suivants illustrent comment utiliser `CreateAlias`.

CLI

AWS CLI

Pour créer un alias de fonction Lambda

L'exemple `create-alias` suivant crée un alias de fonction Lambda nommé `LIVE` qui pointe vers la version 1 de la fonction Lambda `my-function`.

```
aws lambda create-alias \
    --function-name my-function \
    --description "alias for live version of function" \
    --function-version 1 \
    --name LIVE
```

Sortie :

```
{
    "FunctionVersion": "1",
    "Name": "LIVE",
    "AliasArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:LIVE",
    "RevisionId": "873282ed-4cd3-4dc8-a069-d0c647e470c6",
    "Description": "alias for live version of function"
}
```

Pour plus d'informations, consultez la section Configuration des alias de fonction AWS Lambda dans le guide du développeur Lambda AWS .

- Pour plus de détails sur l'API, voir CreateAliasla section Référence des AWS CLI commandes.

PowerShell

Outils pour PowerShell V4

Exemple 1 : cet exemple crée un alias Lambda pour une version et une configuration de routage spécifiées afin de déterminer le pourcentage de requêtes d'invocation reçues.

```
New-LMAlias -FunctionName "MylambdaFunction123" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6} -Description "Alias
 for version 4" -FunctionVersion 4 -Name "PowershellAlias"
```

- Pour plus de détails sur l'API, reportez-vous CreateAliasà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : cet exemple crée un alias Lambda pour une version et une configuration de routage spécifiées afin de déterminer le pourcentage de requêtes d'invocation reçues.

```
New-LMAlias -FunctionName "MylambdaFunction123" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6} -Description "Alias
 for version 4" -FunctionVersion 4 -Name "PowershellAlias"
```

- Pour plus de détails sur l'API, reportez-vous CreateAliasà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation **`CreateFunction`** avec un AWS SDK ou une CLI

Les exemples de code suivants illustrent comment utiliser `CreateFunction`.

Les exemples d'actions sont des extraits de code de programmes de plus grande envergure et doivent être exécutés en contexte. Vous pouvez voir cette action en contexte dans l'exemple de code suivant :

- Principes de base

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    /// <summary>
```

```
    /// Creates a new Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the function.</param>
    /// <param name="s3Bucket">The Amazon Simple Storage Service (Amazon S3)
    /// bucket where the zip file containing the code is located.</param>
    /// <param name="s3Key">The Amazon S3 key of the zip file.</param>
    /// <param name="role">The Amazon Resource Name (ARN) of a role with the
    /// appropriate Lambda permissions.</param>
    /// <param name="handler">The name of the handler function.</param>
    /// <returns>The Amazon Resource Name (ARN) of the newly created
    /// Lambda function.</returns>
    public async Task<string> CreateLambdaFunctionAsync(
        string functionName,
        string s3Bucket,
        string s3Key,
        string role,
        string handler)
    {
        // Defines the location for the function code.
        // S3Bucket - The S3 bucket where the file containing
        //            the source code is stored.
        // S3Key    - The name of the file containing the code.
        var functionCode = new FunctionCode
        {
            S3Bucket = s3Bucket,
            S3Key = s3Key,
        };

        var createFunctionRequest = new CreateFunctionRequest
        {
            FunctionName = functionName,
            Description = "Created by the Lambda .NET API",
            Code = functionCode,
            Handler = handler,
            Runtime = Runtime.Dotnet6,
            Role = role,
        };

        var reponse = await
_lambdaService.CreateFunctionAsync(createFunctionRequest);
        return reponse.FunctionArn;
    }
```

- Pour plus de détails sur l'API, reportez-vous CreateFunctionà la section Référence des AWS SDK pour .NET API.

C++

SDK pour C++

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
 (overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::CreateFunctionRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetDescription(LAMBDA_DESCRIPTION); // Optional.
#if USE_CPP_LAMBDA_FUNCTION
        request.SetRuntime(Aws::Lambda::Model::Runtime::provided_al2);
        request.SetTimeout(15);
        request.SetMemorySize(128);

        // Assume the AWS Lambda function was built in Docker with same
 architecture
        // as this code.
#if  defined(__x86_64__)
        request.SetArchitectures({Aws::Lambda::Model::Architecture::x86_64});
#elif defined(__aarch64__)
        request.SetArchitectures({Aws::Lambda::Model::Architecture::arm64});
#else
#error "Unimplemented architecture"
#endif // defined(architecture)
#else
```

```
        request.SetRuntime(Aws::Lambda::Model::Runtime::python3_9);
#endif
        request.SetRole(roleArn);
        request.SetHandler(LAMBDA_HANDLER_NAME);
        request.SetPublish(true);
        Aws::Lambda::Model::FunctionCode code;
        std::ifstream ifstream(INCREMENT_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteIamRole(clientConfig);
            return false;
        }

        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();

        code.SetZipFile(Aws::Utils::ByteBuffer((unsigned char *)
 buffer.str().c_str(),
                                               buffer.str().length()));
        request.SetCode(code);

        Aws::Lambda::Model::CreateFunctionOutcome outcome =
 client.CreateFunction(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda function was successfully created. " <<
 seconds
                      << " seconds elapsed." << std::endl;
            break;
        }

        else {
            std::cerr << "Error with CreateFunction. "
                      << outcome.GetError().GetMessage()
```

```
                        << std::endl;
            deleteIamRole(clientConfig);
            return false;
        }
```

- Pour plus de détails sur l'API, reportez-vous CreateFunctionà la section Référence des AWS SDK pour C++ API.

CLI

AWS CLI

Pour créer une fonction Lambda

L'exemple `create-function` suivant crée une fonction Lambda nommée `my-function`.

```
aws lambda create-function \
    --function-name my-function \
    --runtime nodejs22.x \
    --zip-file fileb://my-function.zip \
    --handler my-function.handler \
    --role arn:aws:iam::123456789012:role/service-role/MyTestFunction-role-
tges6bf4
```

Contenu de `my-function.zip` :

```
This file is a deployment package that contains your function code and any
 dependencies.
```

Sortie :

```
{
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "PFn4S+er27qk+UuZSTKEQfNKG/XNn7QJs90mJgq6oH8=",
    "FunctionName": "my-function",
    "CodeSize": 308,
    "RevisionId": "873282ed-4cd3-4dc8-a069-d0c647e470c6",
    "MemorySize": 128,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
```

```
    "Version": "$LATEST",
    "Role": "arn:aws:iam::123456789012:role/service-role/MyTestFunction-role-
zgur6bf4",
    "Timeout": 3,
    "LastModified": "2025-10-14T22:26:11.234+0000",
    "Handler": "my-function.handler",
    "Runtime": "nodejs22.x",
    "Description": ""
}
```

Pour plus d'informations, consultez Configurer la mémoire de fonction Lambda dans le Guide du développeur AWS Lambda.

- Pour plus de détails sur l'API, voir CreateFunctionla section Référence des AWS CLI commandes.

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
```

```
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// CreateFunction creates a new Lambda function from code contained in the
 zipPackage
// buffer. The specified handlerName must match the name of the file and function
// contained in the uploaded code. The role specified by iamRoleArn is assumed by
// Lambda and grants specific permissions.
// When the function already exists, types.StateActive is returned.
// When the function is created, a lambda.FunctionActiveV2Waiter is used to wait
 until the
// function is active.
func (wrapper FunctionWrapper) CreateFunction(ctx context.Context, functionName
 string, handlerName string,
 iamRoleArn *string, zipPackage *bytes.Buffer) types.State {
 var state types.State
 _, err := wrapper.LambdaClient.CreateFunction(ctx, &lambda.CreateFunctionInput{
  Code:         &types.FunctionCode{ZipFile: zipPackage.Bytes()},
  FunctionName: aws.String(functionName),
  Role:         iamRoleArn,
  Handler:      aws.String(handlerName),
  Publish:      true,
  Runtime:      types.RuntimePython39,
 })
 if err != nil {
  var resConflict *types.ResourceConflictException
  if errors.As(err, &resConflict) {
   log.Printf("Function %v already exists.\n", functionName)
   state = types.StateActive
  } else {
   log.Panicf("Couldn't create function %v. Here's why: %v\n", functionName, err)
  }
 } else {
  waiter := lambda.NewFunctionActiveV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(ctx, &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
 functionName, err)
  } else {
   state = funcOutput.Configuration.State
```

```
  }
 }
 return state
}
```

- Pour plus de détails sur l'API, reportez-vous CreateFunctionà la section Référence des AWS SDK pour Go API.

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    /**
     * Creates a new Lambda function in AWS using the AWS Lambda Java API.
     *
     * @param awsLambda    the AWS Lambda client used to interact with the AWS
Lambda service
     * @param functionName the name of the Lambda function to create
     * @param key          the S3 key of the function code
     * @param bucketName   the name of the S3 bucket containing the function code
     * @param role         the IAM role to assign to the Lambda function
     * @param handler      the fully qualified class name of the function handler
     * @return the Amazon Resource Name (ARN) of the created Lambda function
     */
    public static String createLambdaFunction(LambdaClient awsLambda,
                                               String functionName,
                                               String key,
                                               String bucketName,
                                               String role,
                                               String handler) {

        try {
            LambdaWaiter waiter = awsLambda.waiter();
```

```
            FunctionCode code = FunctionCode.builder()
                .s3Key(key)
                .s3Bucket(bucketName)
                .build();

            CreateFunctionRequest functionRequest =
CreateFunctionRequest.builder()
                .functionName(functionName)
                .description("Created by the Lambda Java API")
                .code(code)
                .handler(handler)
                .runtime(Runtime.JAVA17)
                .role(role)
                .build();

            // Create a Lambda function using a waiter
            CreateFunctionResponse functionResponse =
awsLambda.createFunction(functionRequest);
            GetFunctionRequest getFunctionRequest = GetFunctionRequest.builder()
                .functionName(functionName)
                .build();
            WaiterResponse<GetFunctionResponse> waiterResponse =
waiter.waitUntilFunctionExists(getFunctionRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            return functionResponse.functionArn();

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return "";
    }
```

- Pour plus de détails sur l'API, reportez-vous CreateFunctionà la section Référence des AWS SDK for Java 2.x API.

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
const createFunction = async (funcName, roleArn) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${funcName}.zip`);

  const command = new CreateFunctionCommand({
    Code: { ZipFile: code },
    FunctionName: funcName,
    Role: roleArn,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};
```

- Pour plus de détails sur l'API, reportez-vous CreateFunctionà la section Référence des AWS SDK pour JavaScript API.

Kotlin

SDK pour Kotlin

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
suspend fun createNewFunction(
    myFunctionName: String,
    s3BucketName: String,
    myS3Key: String,
    myHandler: String,
    myRole: String,
): String? {
    val functionCode =
        FunctionCode {
            s3Bucket = s3BucketName
            s3Key = myS3Key
        }

    val request =
        CreateFunctionRequest {
            functionName = myFunctionName
            code = functionCode
            description = "Created by the Lambda Kotlin API"
            handler = myHandler
            role = myRole
            runtime = Runtime.Java17
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        val functionResponse = awsLambda.createFunction(request)
        awsLambda.waitUntilFunctionActive {
            functionName = myFunctionName
        }
        return functionResponse.functionArn
    }
}
```

- Pour plus de détails sur l'API, consultez CreateFunctionla section AWS SDK pour la référence de l'API Kotlin.

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    public function createFunction($functionName, $role, $bucketName, $handler)
    {
        //This assumes the Lambda function is in an S3 bucket.
        return $this->customWaiter(function () use ($functionName, $role,
$bucketName, $handler) {
            return $this->lambdaClient->createFunction([
                'Code' => [
                    'S3Bucket' => $bucketName,
                    'S3Key' => $functionName,
                ],
                'FunctionName' => $functionName,
                'Role' => $role['Arn'],
                'Runtime' => 'python3.9',
                'Handler' => "$handler.lambda_handler",
            ]);
        });
    }
```

- Pour plus de détails sur l'API, reportez-vous CreateFunctionà la section Référence des AWS SDK pour PHP API.

PowerShell

Outils pour PowerShell V4

Exemple 1 : Cet exemple crée une nouvelle fonction C# (dotnetcore1.0 runtime) nommée dans MyFunction AWS Lambda, fournissant les fichiers binaires compilés pour la fonction à partir d'un fichier zip sur le système de fichiers local (des chemins relatifs ou absolus peuvent être utilisés). Les fonctions Lambda C# spécifient le gestionnaire de la fonction en utilisant la désignation : :Namespace. AssemblyName ClassName: :MethodName. Vous devez remplacer

les parties du nom de l'assemblage (sans le suffixe .dll), de l'espace de noms, du nom de classe et du nom de méthode de la spécification du gestionnaire de manière appropriée. La nouvelle fonction aura les variables d'environnement « envvar1 » et « envvar2 » configurées à partir des valeurs fournies.

```
Publish-LMFunction -Description "My C# Lambda Function" `
        -FunctionName MyFunction `
        -ZipFilename .\MyFunctionBinaries.zip `
        -Handler "AssemblyName::Namespace.ClassName::MethodName" `
        -Role "arn:aws:iam::123456789012:role/LambdaFullExecRole" `
        -Runtime dotnetcore1.0 `
        -Environment_Variable @{ "envvar1"="value";"envvar2"="value" }
```

Sortie :

```
CodeSha256       : /NgBMd...gq71I=
CodeSize         : 214784
DeadLetterConfig :
Description      : My C# Lambda Function
Environment      : Amazon.Lambda.Model.EnvironmentResponse
FunctionArn      : arn:aws:lambda:us-west-2:123456789012:function:ToUpper
FunctionName     : MyFunction
Handler          : AssemblyName::Namespace.ClassName::MethodName
KMSKeyArn        :
LastModified     : 2016-12-29T23:50:14.207+0000
MemorySize       : 128
Role             : arn:aws:iam::123456789012:role/LambdaFullExecRole
Runtime          : dotnetcore1.0
Timeout          : 3
Version          : $LATEST
VpcConfig        :
```

Exemple 2 : cet exemple est similaire au précédent, sauf que les fichiers binaires des fonctions sont d'abord chargés dans un compartiment Amazon S3 (qui doit se trouver dans la même région que la fonction Lambda prévue) et que l'objet S3 obtenu est ensuite référencé lors de la création de la fonction.

```
Write-S3Object -BucketName amzn-s3-demo-bucket -Key MyFunctionBinaries.zip -
File .\MyFunctionBinaries.zip
Publish-LMFunction -Description "My C# Lambda Function" `
        -FunctionName MyFunction `
```

```
        -BucketName amzn-s3-demo-bucket `
        -Key MyFunctionBinaries.zip `
        -Handler "AssemblyName::Namespace.ClassName::MethodName" `
        -Role "arn:aws:iam::123456789012:role/LambdaFullExecRole" `
        -Runtime dotnetcore1.0 `
        -Environment_Variable @{ "envvar1"="value";"envvar2"="value" }
```

- Pour plus de détails sur l'API, reportez-vous CreateFunctionà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : Cet exemple crée une nouvelle fonction C# (dotnetcore1.0 runtime) nommée dans MyFunction AWS Lambda, fournissant les fichiers binaires compilés pour la fonction à partir d'un fichier zip sur le système de fichiers local (des chemins relatifs ou absolus peuvent être utilisés). Les fonctions Lambda C# spécifient le gestionnaire de la fonction en utilisant la désignation : :Namespace. AssemblyName ClassName: :MethodName. Vous devez remplacer les parties du nom de l'assemblage (sans le suffixe .dll), de l'espace de noms, du nom de classe et du nom de méthode de la spécification du gestionnaire de manière appropriée. La nouvelle fonction aura les variables d'environnement « envvar1 » et « envvar2 » configurées à partir des valeurs fournies.

```
Publish-LMFunction -Description "My C# Lambda Function" `
        -FunctionName MyFunction `
        -ZipFilename .\MyFunctionBinaries.zip `
        -Handler "AssemblyName::Namespace.ClassName::MethodName" `
        -Role "arn:aws:iam::123456789012:role/LambdaFullExecRole" `
        -Runtime dotnetcore1.0 `
        -Environment_Variable @{ "envvar1"="value";"envvar2"="value" }
```

Sortie :

```
CodeSha256       : /NgBMd...gq71I=
CodeSize         : 214784
DeadLetterConfig :
Description      : My C# Lambda Function
Environment      : Amazon.Lambda.Model.EnvironmentResponse
FunctionArn      : arn:aws:lambda:us-west-2:123456789012:function:ToUpper
FunctionName     : MyFunction
Handler          : AssemblyName::Namespace.ClassName::MethodName
KMSKeyArn        :
LastModified     : 2016-12-29T23:50:14.207+0000
```

```
MemorySize       : 128
Role             : arn:aws:iam::123456789012:role/LambdaFullExecRole
Runtime          : dotnetcore1.0
Timeout          : 3
Version          : $LATEST
VpcConfig        :
```

Exemple 2 : cet exemple est similaire au précédent, sauf que les fichiers binaires des fonctions sont d'abord chargés dans un compartiment Amazon S3 (qui doit se trouver dans la même région que la fonction Lambda prévue) et que l'objet S3 obtenu est ensuite référencé lors de la création de la fonction.

```
Write-S3Object -BucketName amzn-s3-demo-bucket -Key MyFunctionBinaries.zip -
File .\MyFunctionBinaries.zip
Publish-LMFunction -Description "My C# Lambda Function" `
        -FunctionName MyFunction `
        -BucketName amzn-s3-demo-bucket `
        -Key MyFunctionBinaries.zip `
        -Handler "AssemblyName::Namespace.ClassName::MethodName" `
        -Role "arn:aws:iam::123456789012:role/LambdaFullExecRole" `
        -Runtime dotnetcore1.0 `
        -Environment_Variable @{ "envvar1"="value";"envvar2"="value" }
```

- Pour plus de détails sur l'API, reportez-vous CreateFunctionà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource
```

```
    def create_function(
        self, function_name, handler_name, iam_role, deployment_package
    ):
        """
        Deploys a Lambda function.

        :param function_name: The name of the Lambda function.
        :param handler_name: The fully qualified name of the handler function.
This
                             must include the file name and the function name.
        :param iam_role: The IAM role to use for the function.
        :param deployment_package: The deployment package that contains the
function
                                   code in .zip format.
        :return: The Amazon Resource Name (ARN) of the newly created function.
        """
        try:
            response = self.lambda_client.create_function(
                FunctionName=function_name,
                Description="AWS Lambda doc example",
                Runtime="python3.9",
                Role=iam_role.arn,
                Handler=handler_name,
                Code={"ZipFile": deployment_package},
                Publish=True,
            )
            function_arn = response["FunctionArn"]
            waiter = self.lambda_client.get_waiter("function_active_v2")
            waiter.wait(FunctionName=function_name)
            logger.info(
                "Created function '%s' with ARN: '%s'.",
                function_name,
                response["FunctionArn"],
            )
        except ClientError:
            logger.error("Couldn't create function %s.", function_name)
            raise
        else:
            return function_arn
```

- Pour plus de détails sur l'API, consultez CreateFunctionle AWS manuel de référence de l'API SDK for Python (Boto3).

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Deploys a Lambda function.
  #
  # @param function_name: The name of the Lambda function.
  # @param handler_name: The fully qualified name of the handler function.
  # @param role_arn: The IAM role to use for the function.
  # @param deployment_package: The deployment package that contains the function
 code in .zip format.
  # @return: The Amazon Resource Name (ARN) of the newly created function.
  def create_function(function_name, handler_name, role_arn, deployment_package)
    response = @lambda_client.create_function({
                                                role: role_arn.to_s,
                                                function_name: function_name,
                                                handler: handler_name,
                                                runtime: 'ruby2.7',
                                                code: {
                                                  zip_file: deployment_package
                                                },
```

```
                                                environment: {
                                                  variables: {
                                                    'LOG_LEVEL' => 'info'
                                                  }
                                                }
                                              })
   @lambda_client.wait_until(:function_active_v2, { function_name:
function_name }) do |w|
     w.max_attempts = 5
     w.delay = 5
   end
   response
 rescue Aws::Lambda::Errors::ServiceException => e
   @logger.error("There was an error creating #{function_name}:\n #{e.message}")
 rescue Aws::Waiters::Errors::WaiterFailed => e
   @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
 end
```

- Pour plus de détails sur l'API, reportez-vous CreateFunctionà la section Référence des AWS SDK pour Ruby API.

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
   /**
    * Create a function, uploading from a zip file.
    */
   pub async fn create_function(&self, zip_file: PathBuf) -> Result<String,
anyhow::Error> {
       let code = self.prepare_function(zip_file, None).await?;

       let key = code.s3_key().unwrap().to_string();
```

```
        let role = self.create_role().await.map_err(|e| anyhow!(e))?;

        info!("Created iam role, waiting 15s for it to become active");
        tokio::time::sleep(Duration::from_secs(15)).await;

        info!("Creating lambda function {}", self.lambda_name);
        let _ = self
            .lambda_client
            .create_function()
            .function_name(self.lambda_name.clone())
            .code(code)
            .role(role.arn())
            .runtime(aws_sdk_lambda::types::Runtime::Providedal2)
            .handler("_unused")
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        self.lambda_client
            .publish_version()
            .function_name(self.lambda_name.clone())
            .send()
            .await?;

        Ok(key)
    }

    /**
     * Upload function code from a path to a zip file.
     * The zip file must have an AL2 Linux-compatible binary called `bootstrap`.
     * The easiest way to create such a zip is to use `cargo lambda build --
output-format Zip`.
     */
    async fn prepare_function(
        &self,
        zip_file: PathBuf,
        key: Option<String>,
    ) -> Result<FunctionCode, anyhow::Error> {
        let body = ByteStream::from_path(zip_file).await?;

        let key = key.unwrap_or_else(|| format!("{}_code", self.lambda_name));
```

```
        info!("Uploading function code to s3://{}/{}", self.bucket, key);
        let _ = self
            .s3_client
            .put_object()
            .bucket(self.bucket.clone())
            .key(key.clone())
            .body(body)
            .send()
            .await?;

        Ok(FunctionCode::builder()
            .s3_bucket(self.bucket.clone())
            .s3_key(key)
            .build())
    }
```

- Pour plus de détails sur l'API, voir CreateFunctionla section de référence de l'API AWS SDK for Rust.

SAP ABAP

Kit SDK pour SAP ABAP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    TRY.
        lo_lmd->createfunction(
            iv_functionname = iv_function_name
            iv_runtime = `python3.9`
            iv_role = iv_role_arn
            iv_handler = iv_handler
            io_code = io_zip_file
            iv_description = 'AWS Lambda code example' ).
        MESSAGE 'Lambda function created.' TYPE 'I'.
      CATCH /aws1/cx_lmdcodesigningcfgno00.
```

```
        MESSAGE 'Code signing configuration does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdcodestorageexcdex.
        MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
      CATCH /aws1/cx_lmdcodeverification00.
        MESSAGE 'Code signature failed one or more validation checks for
signature mismatch or expiration.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvalidcodesigex.
        MESSAGE 'Code signature failed the integrity check.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdresourceconflictex.
        MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
      CATCH /aws1/cx_lmdresourcenotfoundex.
        MESSAGE 'The requested resource does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- Pour plus de détails sur l'API, reportez-vous CreateFunctionà la section de référence du AWS SDK pour l'API SAP ABAP.

Swift

Kit SDK pour Swift

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
import AWSClientRuntime
import AWSLambda
import Foundation

        do {
```

```
            // Read the Zip archive containing the AWS Lambda function.

            let zipUrl = URL(fileURLWithPath: path)
            let zipData = try Data(contentsOf: zipUrl)

            // Create the AWS Lambda function that runs the specified code,
            // using the name given on the command line. The Lambda function
            // will run using the Amazon Linux 2 runtime.

            _ = try await lambdaClient.createFunction(
                input: CreateFunctionInput(
                    code: LambdaClientTypes.FunctionCode(zipFile: zipData),
                    functionName: functionName,
                    handler: "handle",
                    role: roleArn,
                    runtime: .providedal2
                )
            )
        } catch {
            print("*** Error creating Lambda function:")
            dump(error)
            return false
        }
```

- Pour plus de détails sur l'API, reportez-vous CreateFunctionà la section AWS SDK pour la référence de l'API Swift.

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation de **`DeleteAlias`** avec une CLI

Les exemples de code suivants illustrent comment utiliser `DeleteAlias`.

CLI

AWS CLI

Pour supprimer un alias de fonction Lambda

L'exemple `delete-alias` suivant supprime l'alias nommé `LIVE` de la fonction Lambda `my-function`.

```
aws lambda delete-alias \
    --function-name my-function \
    --name LIVE
```

Cette commande ne produit aucune sortie.

Pour plus d'informations, consultez la section Configuration des alias de fonction AWS Lambda dans le guide du développeur Lambda AWS .

- Pour plus de détails sur l'API, voir DeleteAliasla section Référence des AWS CLI commandes.

PowerShell

Outils pour PowerShell V4

Exemple 1 : cet exemple supprime l'alias de la fonction Lambda mentionnée dans la commande.

```
Remove-LMAlias -FunctionName "MylambdaFunction123" -Name "NewAlias"
```

- Pour plus de détails sur l'API, reportez-vous DeleteAliasà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : cet exemple supprime l'alias de la fonction Lambda mentionnée dans la commande.

```
Remove-LMAlias -FunctionName "MylambdaFunction123" -Name "NewAlias"
```

- Pour plus de détails sur l'API, reportez-vous DeleteAliasà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation **`DeleteFunction`** avec un AWS SDK ou une CLI

Les exemples de code suivants illustrent comment utiliser `DeleteFunction`.

Les exemples d'actions sont des extraits de code de programmes de plus grande envergure et doivent être exécutés en contexte. Vous pouvez voir cette action en contexte dans l'exemple de code suivant :

- Principes de base

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    /// <summary>
    /// Delete an AWS Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// delete.</param>
    /// <returns>A Boolean value that indicates the success of the action.</
returns>
    public async Task<bool> DeleteFunctionAsync(string functionName)
    {
        var request = new DeleteFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.DeleteFunctionAsync(request);

        // A return value of NoContent means that the request was processed.
        // In this case, the function was deleted, and the return value
        // is intentionally blank.
        return response.HttpStatusCode == System.Net.HttpStatusCode.NoContent;
    }
```

- Pour plus de détails sur l'API, reportez-vous DeleteFunctionà la section Référence des AWS SDK pour .NET API.

C++

SDK pour C++

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

    Aws::Lambda::Model::DeleteFunctionRequest request;
    request.SetFunctionName(LAMBDA_NAME);

    Aws::Lambda::Model::DeleteFunctionOutcome outcome = client.DeleteFunction(
            request);

    if (outcome.IsSuccess()) {
        std::cout << "The lambda function was successfully deleted." <<
std::endl;
    }
    else {
        std::cerr << "Error with Lambda::DeleteFunction. "
                  << outcome.GetError().GetMessage()
                  << std::endl;
    }
```

- Pour plus de détails sur l'API, reportez-vous DeleteFunctionà la section Référence des AWS SDK pour C++ API.

CLI

AWS CLI

Exemple 1 : pour supprimer une fonction Lambda par nom de fonction

L'exemple `delete-function` suivant supprime la fonction Lambda `my-function` nommée en spécifiant le nom de la fonction.

```
aws lambda delete-function \
    --function-name my-function
```

Cette commande ne produit aucune sortie.

Exemple 2 : pour supprimer une fonction Lambda par ARN de fonction

L'exemple `delete-function` suivant supprime la fonction Lambda nommée `my-function` en spécifiant l'ARN de la fonction.

```
aws lambda delete-function \
    --function-name arn:aws:lambda:us-west-2:123456789012:function:my-function
```

Cette commande ne produit aucune sortie.

Exemple 3 : pour supprimer une fonction Lambda par ARN partiel de fonction

L'exemple `delete-function` suivant supprime la fonction Lambda nommée `my-function` en spécifiant l'ARN partiel de la fonction.

```
aws lambda delete-function \
    --function-name 123456789012:function:my-function
```

Cette commande ne produit aucune sortie.

Pour plus d'informations, consultez Configuration des options de fonction Lambda AWS dans le Guide du développeur AWS .

- Pour plus de détails sur l'API, voir DeleteFunctionla section Référence des AWS CLI commandes.

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// DeleteFunction deletes the Lambda function specified by functionName.
func (wrapper FunctionWrapper) DeleteFunction(ctx context.Context, functionName
 string) {
 _, err := wrapper.LambdaClient.DeleteFunction(ctx, &lambda.DeleteFunctionInput{
  FunctionName: aws.String(functionName),
 })
```

```
 if err != nil {
  log.Panicf("Couldn't delete function %v. Here's why: %v\n", functionName, err)
 }
}
```

- Pour plus de détails sur l'API, reportez-vous DeleteFunctionà la section Référence des AWS SDK pour Go API.

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
   /**
    * Deletes an AWS Lambda function.
    *
    * @param awsLambda     an instance of the {@link LambdaClient} class, which
is used to interact with the AWS Lambda service
    * @param functionName  the name of the Lambda function to be deleted
    *
    * @throws LambdaException if an error occurs while deleting the Lambda
function
    */
   public static void deleteLambdaFunction(LambdaClient awsLambda, String
functionName) {
       try {
           DeleteFunctionRequest request = DeleteFunctionRequest.builder()
               .functionName(functionName)
               .build();

           awsLambda.deleteFunction(request);
           System.out.println("The " + functionName + " function was deleted");

       } catch (LambdaException e) {
```

```
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

- Pour plus de détails sur l'API, reportez-vous DeleteFunctionà la section Référence des AWS SDK for Java 2.x API.

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
/**
 * @param {string} funcName
 */
const deleteFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new DeleteFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- Pour plus de détails sur l'API, reportez-vous DeleteFunctionà la section Référence des AWS SDK pour JavaScript API.

Kotlin

SDK pour Kotlin

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
suspend fun delLambdaFunction(myFunctionName: String) {
    val request =
        DeleteFunctionRequest {
            functionName = myFunctionName
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        awsLambda.deleteFunction(request)
        println("$myFunctionName was deleted")
    }
}
```

- Pour plus de détails sur l'API, consultez DeleteFunctionla section AWS SDK pour la référence de l'API Kotlin.

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    public function deleteFunction($functionName)
    {
        return $this->lambdaClient->deleteFunction([
            'FunctionName' => $functionName,
```

```
        ]);
    }
```

- Pour plus de détails sur l'API, reportez-vous DeleteFunctionà la section Référence des AWS SDK pour PHP API.

PowerShell

Outils pour PowerShell V4

Exemple 1 : cet exemple supprime une version spécifique d'une fonction Lambda

```
Remove-LMFunction -FunctionName "MylambdaFunction123" -Qualifier '3'
```

- Pour plus de détails sur l'API, reportez-vous DeleteFunctionà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : cet exemple supprime une version spécifique d'une fonction Lambda

```
Remove-LMFunction -FunctionName "MylambdaFunction123" -Qualifier '3'
```

- Pour plus de détails sur l'API, reportez-vous DeleteFunctionà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
```

```
        self.iam_resource = iam_resource


    def delete_function(self, function_name):
        """
        Deletes a Lambda function.

        :param function_name: The name of the function to delete.
        """
        try:
            self.lambda_client.delete_function(FunctionName=function_name)
        except ClientError:
            logger.exception("Couldn't delete function %s.", function_name)
            raise
```

- Pour plus de détails sur l'API, consultez DeleteFunctionle AWS manuel de référence de l'API SDK for Python (Boto3).

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end
```

```
  # Deletes a Lambda function.
  # @param function_name: The name of the function to delete.
  def delete_function(function_name)
    print "Deleting function: #{function_name}..."
    @lambda_client.delete_function(
      function_name: function_name
    )
    print 'Done!'.green
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error deleting #{function_name}:\n #{e.message}")
  end
```

- Pour plus de détails sur l'API, reportez-vous DeleteFunctionà la section Référence des AWS SDK pour Ruby API.

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
   /** Delete a function and its role, and if possible or necessary, its
associated code object and bucket. */
   pub async fn delete_function(
       &self,
       location: Option<String>,
   ) -> (
       Result<DeleteFunctionOutput, anyhow::Error>,
       Result<DeleteRoleOutput, anyhow::Error>,
       Option<Result<DeleteObjectOutput, anyhow::Error>>,
   ) {
       info!("Deleting lambda function {}", self.lambda_name);
       let delete_function = self
           .lambda_client
           .delete_function()
           .function_name(self.lambda_name.clone())
```

```
            .send()
            .await
            .map_err(anyhow::Error::from);

        info!("Deleting iam role {}", self.role_name);
        let delete_role = self
            .iam_client
            .delete_role()
            .role_name(self.role_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from);

        let delete_object: Option<Result<DeleteObjectOutput, anyhow::Error>> =
            if let Some(location) = location {
                info!("Deleting object {location}");
                Some(
                    self.s3_client
                        .delete_object()
                        .bucket(self.bucket.clone())
                        .key(location)
                        .send()
                        .await
                        .map_err(anyhow::Error::from),
                )
            } else {
                info!(?location, "Skipping delete object");
                None
            };

        (delete_function, delete_role, delete_object)
    }
```

- Pour plus de détails sur l'API, voir DeleteFunctionla section de référence de l'API AWS SDK for Rust.

SAP ABAP

Kit SDK pour SAP ABAP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    TRY.
        lo_lmd->deletefunction( iv_functionname = iv_function_name ).
        MESSAGE 'Lambda function deleted.' TYPE 'I'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdresourceconflictex.
        MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
      CATCH /aws1/cx_lmdresourcenotfoundex.
        MESSAGE 'The requested resource does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- Pour plus de détails sur l'API, reportez-vous DeleteFunctionà la section de référence du AWS SDK pour l'API SAP ABAP.

Swift

Kit SDK pour Swift

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
import AWSClientRuntime
import AWSLambda
import Foundation

        do {
            _ = try await lambdaClient.deleteFunction(
                input: DeleteFunctionInput(
                    functionName: "lambda-basics-function"
                )
            )
        } catch {
            print("Error: Unable to delete the function.")
        }
```

- Pour plus de détails sur l'API, reportez-vous DeleteFunctionà la section AWS SDK pour la référence de l'API Swift.

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation de **`DeleteFunctionConcurrency`** avec une CLI

Les exemples de code suivants illustrent comment utiliser `DeleteFunctionConcurrency`.

CLI

AWS CLI

Pour supprimer une limite d'exécution simultanée d'une fonction

L'exemple `delete-function-concurrency` suivant supprime la limite d'exécution simultanée réservée de la fonction `my-function`.

```
aws lambda delete-function-concurrency \
    --function-name  my-function
```

Cette commande ne produit aucune sortie.

Pour plus d'informations, consultez Réservation de simultanéité pour une fonction Lambda dans le Guide du développeur AWS Lambda.

- Pour plus de détails sur l'API, voir DeleteFunctionConcurrencyla section Référence des AWS CLI commandes.

PowerShell

Outils pour PowerShell V4

Exemple 1 : cet exemple supprime la simultanéité des fonctions de la fonction Lambda.

```
Remove-LMFunctionConcurrency -FunctionName "MylambdaFunction123"
```

- Pour plus de détails sur l'API, reportez-vous DeleteFunctionConcurrencyà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : cet exemple supprime la simultanéité des fonctions de la fonction Lambda.

```
Remove-LMFunctionConcurrency -FunctionName "MylambdaFunction123"
```

- Pour plus de détails sur l'API, reportez-vous DeleteFunctionConcurrencyà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation de **`DeleteProvisionedConcurrencyConfig`** avec une CLI

Les exemples de code suivants illustrent comment utiliser `DeleteProvisionedConcurrencyConfig`.

CLI

AWS CLI

Pour supprimer une configuration de simultanéité allouée

L'exemple `delete-provisioned-concurrency-config` suivant supprime la configuration de simultanéité allouée pour l'alias `GREEN` de la fonction spécifiée.

```
aws lambda delete-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier GREEN
```

- Pour plus de détails sur l'API, voir DeleteProvisionedConcurrencyConfigla section Référence des AWS CLI commandes.

PowerShell

Outils pour PowerShell V4

Exemple 1 : cet exemple supprime la configuration de simultanéité allouée pour un alias spécifique.

```
Remove-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

- Pour plus de détails sur l'API, reportez-vous DeleteProvisionedConcurrencyConfigà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : cet exemple supprime la configuration de simultanéité allouée pour un alias spécifique.

```
Remove-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

- Pour plus de détails sur l'API, reportez-vous DeleteProvisionedConcurrencyConfigà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation de **`GetAccountSettings`** avec une CLI

Les exemples de code suivants illustrent comment utiliser `GetAccountSettings`.

CLI

AWS CLI

Pour récupérer les informations relatives à votre compte dans une AWS région

L'exemple `get-account-settings` suivant affiche les limites Lambda et les informations d'utilisation de votre compte.

```
aws lambda get-account-settings
```

Sortie :

```
{
    "AccountLimit": {
       "CodeSizeUnzipped": 262144000,
       "UnreservedConcurrentExecutions": 1000,
       "ConcurrentExecutions": 1000,
       "CodeSizeZipped": 52428800,
       "TotalCodeSize": 80530636800
    },
    "AccountUsage": {
       "FunctionCount": 4,
       "TotalCodeSize": 9426
    }
}
```

Pour plus d'informations, consultez Limites AWS Lambda dans le Guide du développeur AWS Lambda.

- Pour plus de détails sur l'API, voir GetAccountSettingsla section Référence des AWS CLI commandes.

PowerShell

Outils pour PowerShell V4

Exemple 1 : cet exemple affiche une comparaison entre la limite et l'utilisation du compte

```
Get-LMAccountSetting | Select-Object
 @{Name="TotalCodeSizeLimit";Expression={$_.AccountLimit.TotalCodeSize}},
 @{Name="TotalCodeSizeUsed";Expression={$_.AccountUsage.TotalCodeSize}}
```

Sortie :

```
TotalCodeSizeLimit TotalCodeSizeUsed
------------------ -----------------
       80530636800          15078795
```

- Pour plus de détails sur l'API, reportez-vous GetAccountSettingsà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : cet exemple affiche une comparaison entre la limite et l'utilisation du compte

```
Get-LMAccountSetting | Select-Object
 @{Name="TotalCodeSizeLimit";Expression={$_.AccountLimit.TotalCodeSize}},
 @{Name="TotalCodeSizeUsed";Expression={$_.AccountUsage.TotalCodeSize}}
```

Sortie :

```
TotalCodeSizeLimit TotalCodeSizeUsed
------------------ -----------------
       80530636800          15078795
```

- Pour plus de détails sur l'API, reportez-vous GetAccountSettingsà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation de **`GetAlias`** avec une CLI

Les exemples de code suivants illustrent comment utiliser `GetAlias`.

CLI

AWS CLI

Pour récupérer les informations relatives à un alias de fonction

L'exemple `get-alias` suivant affiche les détails de l'alias nommé `LIVE` de la fonction Lambda `my-function`.

```
aws lambda get-alias \
    --function-name my-function \
    --name LIVE
```

Sortie :

```
{
    "FunctionVersion": "3",
    "Name": "LIVE",
    "AliasArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:LIVE",
    "RevisionId": "594f41fb-b85f-4c20-95c7-6ca5f2a92c93",
    "Description": "alias for live version of function"
}
```

Pour plus d'informations, consultez la section Configuration des alias de fonction AWS Lambda dans le guide du développeur Lambda AWS .

- Pour plus de détails sur l'API, voir GetAliasla section Référence des AWS CLI commandes.

PowerShell

Outils pour PowerShell V4

Exemple 1 : cet exemple récupère les poids de la configuration de routage pour un alias de fonction Lambda spécifique.

```
Get-LMAlias -FunctionName "MylambdaFunction123" -Name "newlabel1" -Select
 RoutingConfig
```

Sortie :

```
AdditionalVersionWeights
------------------------
{[1, 0.6]}
```

- Pour plus de détails sur l'API, reportez-vous GetAliasà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : cet exemple récupère les poids de la configuration de routage pour un alias de fonction Lambda spécifique.

```
Get-LMAlias -FunctionName "MylambdaFunction123" -Name "newlabel1" -Select
 RoutingConfig
```

Sortie :

```
AdditionalVersionWeights
------------------------
{[1, 0.6]}
```

- Pour plus de détails sur l'API, reportez-vous GetAliasà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation **`GetFunction`** avec un AWS SDK ou une CLI

Les exemples de code suivants illustrent comment utiliser `GetFunction`.

Les exemples d'actions sont des extraits de code de programmes de plus grande envergure et doivent être exécutés en contexte. Vous pouvez voir cette action en contexte dans l'exemple de code suivant :

- Principes de base

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    /// <summary>
    /// Gets information about a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function for
    /// which to retrieve information.</param>
    /// <returns>Async Task.</returns>
    public async Task<FunctionConfiguration> GetFunctionAsync(string
functionName)
    {
        var functionRequest = new GetFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.GetFunctionAsync(functionRequest);
        return response.Configuration;
    }
```

- Pour plus de détails sur l'API, reportez-vous GetFunctionà la section Référence des AWS SDK pour .NET API.

C++

SDK pour C++

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
       Aws::Client::ClientConfiguration clientConfig;
       // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
       // clientConfig.region = "us-east-1";

   Aws::Lambda::LambdaClient client(clientConfig);

       Aws::Lambda::Model::GetFunctionRequest request;
       request.SetFunctionName(functionName);

       Aws::Lambda::Model::GetFunctionOutcome outcome =
client.GetFunction(request);

       if (outcome.IsSuccess()) {
           std::cout << "Function retrieve.\n" <<

outcome.GetResult().GetConfiguration().Jsonize().View().WriteReadable()
                     << std::endl;
       }
       else {
           std::cerr << "Error with Lambda::GetFunction. "
                     << outcome.GetError().GetMessage()
                     << std::endl;
       }
```

- Pour plus de détails sur l'API, reportez-vous GetFunctionà la section Référence des AWS SDK pour C++ API.

CLI

AWS CLI

Pour récupérer les informations sur une fonction

L'exemple `get-function` suivant affiche des informations sur la fonction `my-function`.

```
aws lambda get-function \
    --function-name  my-function
```

Sortie :

```
{
    "Concurrency": {
        "ReservedConcurrentExecutions": 100
    },
    "Code": {
        "RepositoryType": "S3",
        "Location": "https://awslambda-us-west-2-tasks.s3.us-
west-2.amazonaws.com/snapshots/123456789012/my-function..."
    },
    "Configuration": {
        "TracingConfig": {
            "Mode": "PassThrough"
        },
        "Version": "$LATEST",
        "CodeSha256": "5tT2qgzYUHoqwR616pZ2dpkn/0J1FrzJmlKidWaaCgk=",
        "FunctionName": "my-function",
        "VpcConfig": {
            "SubnetIds": [],
            "VpcId": "",
            "SecurityGroupIds": []
        },
        "MemorySize": 128,
        "RevisionId": "28f0fb31-5c5c-43d3-8955-03e76c5c1075",
        "CodeSize": 304,
        "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function",
        "Handler": "index.handler",
        "Role": "arn:aws:iam::123456789012:role/service-role/helloWorldPython-
role-uy3l9qyq",
        "Timeout": 3,
```

```
        "LastModified": "2025-09-24T18:20:35.054+0000",
        "Runtime": "nodejs22.x",
        "Description": ""
    }
}
```

Pour plus d'informations, consultez Configurer la mémoire de fonction Lambda dans le Guide du développeur AWS Lambda.

- Pour plus de détails sur l'API, voir GetFunctionla section Référence des AWS CLI commandes.

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}
```

```
// GetFunction gets data about the Lambda function specified by functionName.
func (wrapper FunctionWrapper) GetFunction(ctx context.Context, functionName
 string) types.State {
 var state types.State
 funcOutput, err := wrapper.LambdaClient.GetFunction(ctx,
 &lambda.GetFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't get function %v. Here's why: %v\n", functionName, err)
 } else {
  state = funcOutput.Configuration.State
 }
 return state
}
```

- Pour plus de détails sur l'API, reportez-vous GetFunctionà la section Référence des AWS SDK pour Go API.

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    /**
     * Retrieves information about an AWS Lambda function.
     *
     * @param awsLambda    an instance of the {@link LambdaClient} class, which
is used to interact with the AWS Lambda service
     * @param functionName the name of the AWS Lambda function to retrieve
information about
```

```
     */
    public static void getFunction(LambdaClient awsLambda, String functionName) {
        try {
            GetFunctionRequest functionRequest = GetFunctionRequest.builder()
                .functionName(functionName)
                .build();

            GetFunctionResponse response =
 awsLambda.getFunction(functionRequest);
            System.out.println("The runtime of this Lambda function is " +
 response.configuration().runtime());

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

- Pour plus de détails sur l'API, reportez-vous GetFunctionà la section Référence des AWS SDK for Java 2.x API.

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
const getFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new GetFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- Pour plus de détails sur l'API, reportez-vous GetFunctionà la section Référence des AWS SDK pour JavaScript API.

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    public function getFunction($functionName)
    {
        return $this->lambdaClient->getFunction([
            'FunctionName' => $functionName,
        ]);
    }
```

- Pour plus de détails sur l'API, reportez-vous GetFunctionà la section Référence des AWS SDK pour PHP API.

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def get_function(self, function_name):
        """
        Gets data about a Lambda function.
```

```
        :param function_name: The name of the function.
        :return: The function data.
        """
        response = None
        try:
            response =
 self.lambda_client.get_function(FunctionName=function_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.info("Function %s does not exist.", function_name)
            else:
                logger.error(
                    "Couldn't get function %s. Here's why: %s: %s",
                    function_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        return response
```

- Pour plus de détails sur l'API, consultez GetFunctionle AWS manuel de référence de l'API SDK for Python (Boto3).

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
```

```
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Gets data about a Lambda function.
  #
  # @param function_name: The name of the function.
  # @return response: The function data, or nil if no such function exists.
  def get_function(function_name)
    @lambda_client.get_function(
      {
        function_name: function_name
      }
    )
  rescue Aws::Lambda::Errors::ResourceNotFoundException => e
    @logger.debug("Could not find function: #{function_name}:\n #{e.message}")
    nil
  end
```

- Pour plus de détails sur l'API, reportez-vous GetFunctionà la section Référence des AWS SDK pour Ruby API.

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    /** Get the Lambda function with this Manager's name. */
    pub async fn get_function(&self) -> Result<GetFunctionOutput, anyhow::Error>
{
        info!("Getting lambda function");
        self.lambda_client
            .get_function()
```

```
            .function_name(self.lambda_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from)
    }
```

- Pour plus de détails sur l'API, voir GetFunctionla section de référence de l'API AWS SDK for Rust.

SAP ABAP

Kit SDK pour SAP ABAP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    TRY.
        oo_result = lo_lmd->getfunction( iv_functionname = iv_function_name ).
    " oo_result is returned for testing purposes. "
        MESSAGE 'Lambda function information retrieved.' TYPE 'I'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- Pour plus de détails sur l'API, reportez-vous GetFunctionà la section de référence du AWS SDK pour l'API SAP ABAP.

Swift

Kit SDK pour Swift

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
import AWSClientRuntime
import AWSLambda
import Foundation

    /// Detect whether or not the AWS Lambda function with the specified name
    /// exists, by requesting its function information.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `LambdaClient` to use.
    ///   - name: The name of the AWS Lambda function to find.
    ///
    /// - Returns: `true` if the Lambda function exists. Otherwise `false`.
    func doesLambdaFunctionExist(lambdaClient: LambdaClient, name: String) async
 -> Bool {
        do {
            _ = try await lambdaClient.getFunction(
                input: GetFunctionInput(functionName: name)
            )
        } catch {
            return false
        }

        return true
    }
```

- Pour plus de détails sur l'API, reportez-vous GetFunctionà la section AWS SDK pour la référence de l'API Swift.

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation de **`GetFunctionConcurrency`** avec une CLI

Les exemples de code suivants illustrent comment utiliser `GetFunctionConcurrency`.

CLI

AWS CLI

Pour afficher le paramètre de simultanéité réservé pour une fonction

L'exemple `get-function-concurrency` suivant récupère le paramètre de simultanéité réservé pour la fonction spécifiée.

```
aws lambda get-function-concurrency \
    --function-name my-function
```

Sortie :

```
{
    "ReservedConcurrentExecutions": 250
}
```

- Pour plus de détails sur l'API, voir GetFunctionConcurrencyla section Référence des AWS CLI commandes.

PowerShell

Outils pour PowerShell V4

Exemple 1 : cet exemple permet d'obtenir la simultanéité réservée pour la fonction Lambda

```
Get-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -Select *
```

Sortie :

```
ReservedConcurrentExecutions
----------------------------
```

```
100
```

- Pour plus de détails sur l'API, reportez-vous GetFunctionConcurrencyà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : cet exemple permet d'obtenir la simultanéité réservée pour la fonction Lambda

```
Get-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -Select *
```

Sortie :

```
ReservedConcurrentExecutions
----------------------------
100
```

- Pour plus de détails sur l'API, reportez-vous GetFunctionConcurrencyà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation de **`GetFunctionConfiguration`** avec une CLI

Les exemples de code suivants illustrent comment utiliser `GetFunctionConfiguration`.

CLI

AWS CLI

Pour récupérer les paramètres d'une fonction Lambda spécifiques à sa version

L'exemple `get-function-configuration` suivant affiche les paramètres de la version 2 de la fonction `my-function`.

```
aws lambda get-function-configuration \
    --function-name  my-function:2
```

Sortie :

```
{
    "FunctionName": "my-function",
    "LastModified": "2019-09-26T20:28:40.438+0000",
    "RevisionId": "e52502d4-9320-4688-9cd6-152a6ab7490d",
    "MemorySize": 256,
    "Version": "2",
    "Role": "arn:aws:iam::123456789012:role/service-role/my-function-role-
uy3l9qyq",
    "Timeout": 3,
    "Runtime": "nodejs10.x",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "5tT2qgzYUHaqwR716pZ2dpkn/0J1FrzJmlKidWoaCgk=",
    "Description": "",
    "VpcConfig": {
        "SubnetIds": [],
        "VpcId": "",
        "SecurityGroupIds": []
    },
    "CodeSize": 304,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:2",
    "Handler": "index.handler"
}
```

Pour plus d'informations, consultez Configuration des options de fonction Lambda AWS dans le Guide du développeur AWS .

- Pour plus de détails sur l'API, voir GetFunctionConfigurationla section Référence des AWS CLI commandes.

PowerShell

Outils pour PowerShell V4

Exemple 1 : cet exemple renvoie la configuration spécifique à la version d'une fonction Lambda.

```
Get-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Qualifier
 "PowershellAlias"
```

Sortie :

```
CodeSha256                  : uWOW0R7z+f0VyLuUg7+/D08hkMFsq0SF4seuyUZJ/R8=
CodeSize                    : 1426
DeadLetterConfig            : Amazon.Lambda.Model.DeadLetterConfig
Description                 : Verson 3 to test Aliases
Environment                 : Amazon.Lambda.Model.EnvironmentResponse
FunctionArn                 : arn:aws:lambda:us-
east-1:123456789012:function:MylambdaFunction123
                              :PowershellAlias
FunctionName                : MylambdaFunction123
Handler                     : lambda_function.launch_instance
KMSKeyArn                   :
LastModified                : 2019-12-25T09:52:59.872+0000
LastUpdateStatus            : Successful
LastUpdateStatusReason      :
LastUpdateStatusReasonCode  :
Layers                      : {}
MasterArn                   :
MemorySize                  : 128
RevisionId                  : 5d7de38b-87f2-4260-8f8a-e87280e10c33
Role                        : arn:aws:iam::123456789012:role/service-role/lambda
Runtime                     : python3.8
State                       : Active
StateReason                 :
StateReasonCode             :
Timeout                     : 600
TracingConfig               : Amazon.Lambda.Model.TracingConfigResponse
Version                     : 4
VpcConfig                   : Amazon.Lambda.Model.VpcConfigDetail
```

- Pour plus de détails sur l'API, reportez-vous GetFunctionConfigurationà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : cet exemple renvoie la configuration spécifique à la version d’une fonction Lambda.

```
Get-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Qualifier
 "PowershellAlias"
```

Sortie :

```
CodeSha256                  : uWOW0R7z+f0VyLuUg7+/D08hkMFsq0SF4seuyUZJ/R8=
CodeSize                    : 1426
DeadLetterConfig            : Amazon.Lambda.Model.DeadLetterConfig
Description                 : Verson 3 to test Aliases
Environment                 : Amazon.Lambda.Model.EnvironmentResponse
FunctionArn                 : arn:aws:lambda:us-
east-1:123456789012:function:MylambdaFunction123
                              :PowershellAlias
FunctionName                : MylambdaFunction123
Handler                     : lambda_function.launch_instance
KMSKeyArn                   :
LastModified                : 2019-12-25T09:52:59.872+0000
LastUpdateStatus            : Successful
LastUpdateStatusReason      :
LastUpdateStatusReasonCode  :
Layers                      : {}
MasterArn                   :
MemorySize                  : 128
RevisionId                  : 5d7de38b-87f2-4260-8f8a-e87280e10c33
Role                        : arn:aws:iam::123456789012:role/service-role/lambda
Runtime                     : python3.8
State                       : Active
StateReason                 :
StateReasonCode             :
Timeout                     : 600
TracingConfig               : Amazon.Lambda.Model.TracingConfigResponse
Version                     : 4
VpcConfig                   : Amazon.Lambda.Model.VpcConfigDetail
```

- Pour plus de détails sur l'API, reportez-vous GetFunctionConfigurationà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation de **`GetPolicy`** avec une CLI

Les exemples de code suivants illustrent comment utiliser `GetPolicy`.

CLI

AWS CLI

Pour récupérer la politique IAM basée sur les ressources d'une fonction, d'une version ou d'un alias

L'exemple `get-policy` suivant affiche les informations de politique relative à la fonction Lambda `my-function`.

```
aws lambda get-policy \
    --function-name my-function
```

Sortie :

```
{
    "Policy": {
        "Version":"2012-10-17",
        "Id":"default",
        "Statement":
        [
            {
                "Sid":"iot-events",
                "Effect":"Allow",
                "Principal": {"Service":"iotevents.amazonaws.com"},
                "Action":"lambda:InvokeFunction",
                "Resource":"arn:aws:lambda:us-west-2:123456789012:function:my-
function"
            }
        ]
    },
    "RevisionId": "93017fc9-59cb-41dc-901b-4845ce4bf668"
}
```

Pour plus d'informations, consultez la section Utilisation de politiques basées sur les ressources pour Lambda AWS dans le guide du développeur AWS Lambda.

- Pour plus de détails sur l'API, voir GetPolicyla section Référence des AWS CLI commandes.

PowerShell

Outils pour PowerShell V4

Exemple 1 : cet exemple affiche la stratégie de fonction de la fonction Lambda

```
Get-LMPolicy -FunctionName test -Select Policy
```

Sortie :

```
{"Version":"2012-10-17",       "Id":"default","Statement":
[{"Sid":"xxxx","Effect":"Allow","Principal":
{"Service":"sns.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lamb
east-1:123456789102:function:test"}]}
```

- Pour plus de détails sur l'API, reportez-vous GetPolicyà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : cet exemple affiche la stratégie de fonction de la fonction Lambda

```
Get-LMPolicy -FunctionName test -Select Policy
```

Sortie :

```
{"Version":"2012-10-17",       "Id":"default","Statement":
[{"Sid":"xxxx","Effect":"Allow","Principal":
{"Service":"sns.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lamb
east-1:123456789102:function:test"}]}
```

- Pour plus de détails sur l'API, reportez-vous GetPolicyà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation de **`GetProvisionedConcurrencyConfig`** avec une CLI

Les exemples de code suivants illustrent comment utiliser `GetProvisionedConcurrencyConfig`.

CLI

AWS CLI

Pour afficher la configuration de la simultanéité allouée

L'exemple `get-provisioned-concurrency-config` suivant affiche les détails de la configuration de simultanéité allouée pour l'alias BLUE de la fonction spécifiée.

```
aws lambda get-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier BLUE
```

Sortie :

```
{
    "RequestedProvisionedConcurrentExecutions": 100,
    "AvailableProvisionedConcurrentExecutions": 100,
    "AllocatedProvisionedConcurrentExecutions": 100,
    "Status": "READY",
    "LastModified": "2019-12-31T20:28:49+0000"
}
```

- Pour plus de détails sur l'API, voir GetProvisionedConcurrencyConfigla section Référence des AWS CLI commandes.

PowerShell

Outils pour PowerShell V4

Exemple 1 : cet exemple obtient la configuration de simultanéité allouée pour l'alias spécifié de la fonction Lambda.

```
C:\>Get-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

Sortie :

```
AllocatedProvisionedConcurrentExecutions : 0
AvailableProvisionedConcurrentExecutions : 0
```

```
LastModified                                : 2020-01-15T03:21:26+0000
RequestedProvisionedConcurrentExecutions : 70
Status                                      : IN_PROGRESS
StatusReason                                :
```

- Pour plus de détails sur l'API, reportez-vous GetProvisionedConcurrencyConfigà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : cet exemple obtient la configuration de simultanéité allouée pour l’alias spécifié de la fonction Lambda.

```
C:\>Get-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

Sortie :

```
AllocatedProvisionedConcurrentExecutions : 0
AvailableProvisionedConcurrentExecutions : 0
LastModified                             : 2020-01-15T03:21:26+0000
RequestedProvisionedConcurrentExecutions : 70
Status                                   : IN_PROGRESS
StatusReason                             :
```

- Pour plus de détails sur l'API, reportez-vous GetProvisionedConcurrencyConfigà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation **`Invoke`** avec un AWS SDK ou une CLI

Les exemples de code suivants illustrent comment utiliser `Invoke`.

Les exemples d’actions sont des extraits de code de programmes de plus grande envergure et doivent être exécutés en contexte. Vous pouvez voir cette action en contexte dans l’exemple de code suivant :

- Principes de base

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    /// <summary>
    /// Invoke a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// invoke.</param
    /// <param name="parameters">The parameter values that will be passed to the
function.</param>
    /// <returns>A System Threading Task.</returns>
    public async Task<string> InvokeFunctionAsync(
        string functionName,
        string parameters)
    {
        var payload = parameters;
        var request = new InvokeRequest
        {
            FunctionName = functionName,
            Payload = payload,
        };

        var response = await _lambdaService.InvokeAsync(request);
        MemoryStream stream = response.Payload;
        string returnValue =
System.Text.Encoding.UTF8.GetString(stream.ToArray());
        return returnValue;
    }
```

- Pour en savoir plus sur l'API, consultez Invoke dans la Référence de l'API AWS SDK pour .NET .

C++

SDK pour C++

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::InvokeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetLogType(logType);
        std::shared_ptr<Aws::IOStream> payload =
Aws::MakeShared<Aws::StringStream>(
                "FunctionTest");
        *payload << jsonPayload.View().WriteReadable();
        request.SetBody(payload);
        request.SetContentType("application/json");
        Aws::Lambda::Model::InvokeOutcome outcome = client.Invoke(request);

        if (outcome.IsSuccess()) {
            invokeResult = std::move(outcome.GetResult());
            result = true;
            break;
        }

        else {
            std::cerr << "Error with Lambda::InvokeRequest. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            break;
        }
```

- Pour en savoir plus sur l'API, consultez Invoke dans la Référence de l'API AWS SDK pour C++ .

CLI

AWS CLI

Exemple 1 : pour invoquer une fonction Lambda de manière synchrone

L'exemple `invoke` suivant invoque la fonction `my-function` de manière synchrone. L'`cli-binary-format`option est obligatoire si vous utilisez la version 2 de la AWS CLI. Pour plus d'informations, veuillez consulter les options de ligne de commande prises en charge par la CLI AWS dans le Guide de l'interface de ligne de commande AWS .

```
aws lambda invoke \
    --function-name my-function \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "name": "Bob" }' \
    response.json
```

Sortie :

```
{
    "ExecutedVersion": "$LATEST",
    "StatusCode": 200
}
```

Pour plus d'informations, consultez Invocation d'une fonction Lambda de manière synchrone dans le Guide du développeur AWS Lambda.

Exemple 2 : pour invoquer une fonction Lambda de manière asynchrone

L'exemple `invoke` suivant invoque la fonction `my-function` de manière asynchrone. L'`cli-binary-format`option est obligatoire si vous utilisez la version 2 de la AWS CLI. Pour plus d'informations, veuillez consulter les options de ligne de commande prises en charge par la CLI AWS dans le Guide de l'interface de ligne de commande AWS .

```
aws lambda invoke \
    --function-name my-function \
```

```
    --invocation-type Event \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "name": "Bob" }' \
    response.json
```

Sortie :

```
{
    "StatusCode": 202
}
```

Pour plus d'informations, consultez Invocation d'une fonction Lambda de manière asynchrone dans le Guide du développeur AWS Lambda.

- Pour plus d'informations sur l'API, consultez Invoke dans la Référence des commandes AWS CLI .

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)
```

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// Invoke invokes the Lambda function specified by functionName, passing the
 parameters
// as a JSON payload. When getLog is true, types.LogTypeTail is specified, which
 tells
// Lambda to include the last few log lines in the returned result.
func (wrapper FunctionWrapper) Invoke(ctx context.Context, functionName string,
 parameters any, getLog bool) *lambda.InvokeOutput {
 logType := types.LogTypeNone
 if getLog {
  logType = types.LogTypeTail
 }
 payload, err := json.Marshal(parameters)
 if err != nil {
  log.Panicf("Couldn't marshal parameters to JSON. Here's why %v\n", err)
 }
 invokeOutput, err := wrapper.LambdaClient.Invoke(ctx, &lambda.InvokeInput{
  FunctionName: aws.String(functionName),
  LogType:      logType,
  Payload:      payload,
 })
 if err != nil {
  log.Panicf("Couldn't invoke function %v. Here's why: %v\n", functionName, err)
 }
 return invokeOutput
}
```

- Pour en savoir plus sur l'API, consultez Invoke dans la Référence de l'API AWS SDK pour Go .

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
   /**
    * Invokes a specific AWS Lambda function.
    *
    * @param awsLambda    an instance of {@link LambdaClient} to interact with
the AWS Lambda service
    * @param functionName the name of the AWS Lambda function to be invoked
    */
   public static void invokeFunction(LambdaClient awsLambda, String
functionName) {
       InvokeResponse res;
       try {
           // Need a SdkBytes instance for the payload.
           JSONObject jsonObj = new JSONObject();
           jsonObj.put("inputValue", "2000");
           String json = jsonObj.toString();
           SdkBytes payload = SdkBytes.fromUtf8String(json);

           InvokeRequest request = InvokeRequest.builder()
               .functionName(functionName)
               .payload(payload)
               .build();

           res = awsLambda.invoke(request);
           String value = res.payload().asUtf8String();
           System.out.println(value);

       } catch (LambdaException e) {
           System.err.println(e.getMessage());
           System.exit(1);
       }
   }
```

- Pour en savoir plus sur l'API, consultez Invoke dans la Référence de l'API AWS SDK for Java 2.x .

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
const invoke = async (funcName, payload) => {
  const client = new LambdaClient({});
  const command = new InvokeCommand({
    FunctionName: funcName,
    Payload: JSON.stringify(payload),
    LogType: LogType.Tail,
  });

  const { Payload, LogResult } = await client.send(command);
  const result = Buffer.from(Payload).toString();
  const logs = Buffer.from(LogResult, "base64").toString();
  return { logs, result };
};
```

- Pour plus d'informations sur l'API, consultez Invoke dans la AWS SDK pour JavaScript Référence d'API.

Kotlin

SDK pour Kotlin

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
suspend fun invokeFunction(functionNameVal: String) {
    val json = """{"inputValue":"1000"}"""
    val byteArray = json.trimIndent().encodeToByteArray()
    val request =
        InvokeRequest {
            functionName = functionNameVal
            logType = LogType.Tail
            payload = byteArray
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val res = awsLambda.invoke(request)
        println("${res.payload?.toString(Charsets.UTF_8)}")
        println("The log result is ${res.logResult}")
    }
}
```

- Pour plus d'informations sur l'API, consultez Invoke dans la AWS Référence d'API du kit SDK pour Kotlin.

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    public function invoke($functionName, $params, $logType = 'None')
    {
        return $this->lambdaClient->invoke([
            'FunctionName' => $functionName,
            'Payload' => json_encode($params),
            'LogType' => $logType,
        ]);
    }
```

- Pour en savoir plus sur l'API, consultez Invoke dans la Référence de l'API AWS SDK pour PHP .

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def invoke_function(self, function_name, function_params, get_log=False):
        """
        Invokes a Lambda function.

        :param function_name: The name of the function to invoke.
        :param function_params: The parameters of the function as a dict. This
 dict
                                is serialized to JSON before it is sent to
 Lambda.
        :param get_log: When true, the last 4 KB of the execution log are
 included in
                        the response.
        :return: The response from the function invocation.
        """
        try:
            response = self.lambda_client.invoke(
                FunctionName=function_name,
                Payload=json.dumps(function_params),
                LogType="Tail" if get_log else "None",
            )
            logger.info("Invoked function %s.", function_name)
        except ClientError:
```

```
            logger.exception("Couldn't invoke function %s.", function_name)
            raise
        return response
```

- Pour plus d'informations sur l'API, consultez Invoke dans la AWS Référence d'API du kit SDK pour Python (Boto3).

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Invokes a Lambda function.
  # @param function_name [String] The name of the function to invoke.
  # @param payload [nil] Payload containing runtime parameters.
  # @return [Object] The response from the function invocation.
  def invoke_function(function_name, payload = nil)
    params = { function_name: function_name }
    params[:payload] = payload unless payload.nil?
    @lambda_client.invoke(params)
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error executing #{function_name}:\n
 #{e.message}")
```

```
  end
```

- Pour en savoir plus sur l'API, consultez Invoke dans la Référence de l'API AWS SDK pour Ruby .

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    /** Invoke the lambda function using calculator InvokeArgs. */
    pub async fn invoke(&self, args: InvokeArgs) -> Result<InvokeOutput,
 anyhow::Error> {
        info!(?args, "Invoking {}", self.lambda_name);
        let payload = serde_json::to_string(&args)?;
        debug!(?payload, "Sending payload");
        self.lambda_client
            .invoke()
            .function_name(self.lambda_name.clone())
            .payload(Blob::new(payload))
            .send()
            .await
            .map_err(anyhow::Error::from)
    }

fn log_invoke_output(invoke: &InvokeOutput, message: &str) {
    if let Some(payload) = invoke.payload().cloned() {
        let payload = String::from_utf8(payload.into_inner());
        info!(?payload, message);
    } else {
        info!("Could not extract payload")
    }
    if let Some(logs) = invoke.log_result() {
        debug!(?logs, "Invoked function logs")
    } else {
```

```
        debug!("Invoked function had no logs")
    }
}
```

- Pour plus d'informations sur l'API, consultez Invoquer dans la Référence d'API du kit SDK AWS pour Rust.

SAP ABAP

Kit SDK pour SAP ABAP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    TRY.
        DATA(lv_json) = /aws1/cl_rt_util=>string_to_xstring(
          `{`  &&
            `"action": "increment",`  &&
            `"number": 10` &&
          `}` ).
        oo_result = lo_lmd->invoke(                 " oo_result is returned for
testing purposes. "
                 iv_functionname = iv_function_name
                 iv_payload = lv_json ).
        MESSAGE 'Lambda function invoked.' TYPE 'I'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvrequestcontex.
        MESSAGE 'Unable to parse request body as JSON.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvalidzipfileex.
        MESSAGE 'The deployment package could not be unzipped.' TYPE 'E'.
      CATCH /aws1/cx_lmdrequesttoolargeex.
        MESSAGE 'Invoke request body JSON input limit was exceeded by the request
payload.' TYPE 'E'.
      CATCH /aws1/cx_lmdresourceconflictex.
        MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
```

```
      CATCH /aws1/cx_lmdresourcenotfoundex.
        MESSAGE 'The requested resource does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
 TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
      CATCH /aws1/cx_lmdunsuppedmediatyp00.
        MESSAGE 'Invoke request body does not have JSON as its content type.'
 TYPE 'E'.
    ENDTRY.
```

- Pour plus de détails sur l'API, consultez Invoke dans la Référence des API du kit AWS SDK pour SAP ABAP.

Swift

Kit SDK pour Swift

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
import AWSClientRuntime
import AWSLambda
import Foundation

    /// Invoke the Lambda function to increment a value.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `IAMClient` to use.
    ///   - number: The number to increment.
    ///
    /// - Throws: `ExampleError.noAnswerReceived`, `ExampleError.invokeError`
    ///
    /// - Returns: An integer number containing the incremented value.
    func invokeIncrement(lambdaClient: LambdaClient, number: Int) async throws ->
 Int {
```

```
        do {
            let incRequest = IncrementRequest(action: "increment", number:
number)
            let incData = try! JSONEncoder().encode(incRequest)

            // Invoke the lambda function.

            let invokeOutput = try await lambdaClient.invoke(
                input: InvokeInput(
                    functionName: "lambda-basics-function",
                    payload: incData
                )
            )

            let response = try! JSONDecoder().decode(Response.self,
from:invokeOutput.payload!)

            guard let answer = response.answer else {
                throw ExampleError.noAnswerReceived
            }
            return answer

        } catch {
            throw ExampleError.invokeError
        }
    }
```

- Pour plus de détails sur l'API, consultez Invoke dans la Référence des API du kit AWS SDK pour Swift.

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation **`ListFunctions`** avec un AWS SDK ou une CLI

Les exemples de code suivants illustrent comment utiliser `ListFunctions`.

Les exemples d'actions sont des extraits de code de programmes de plus grande envergure et doivent être exécutés en contexte. Vous pouvez voir cette action en contexte dans l'exemple de code suivant :

- Principes de base

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
/// <summary>
/// Get a list of Lambda functions.
/// </summary>
/// <returns>A list of FunctionConfiguration objects.</returns>
public async Task<List<FunctionConfiguration>> ListFunctionsAsync()
{
    var functionList = new List<FunctionConfiguration>();

    var functionPaginator =
        _lambdaService.Paginators.ListFunctions(new ListFunctionsRequest());
    await foreach (var function in functionPaginator.Functions)
    {
        functionList.Add(function);
    }

    return functionList;
}
```

- Pour plus de détails sur l'API, reportez-vous ListFunctionsà la section Référence des AWS SDK pour .NET API.

C++

SDK pour C++

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
       Aws::Client::ClientConfiguration clientConfig;
       // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
       // clientConfig.region = "us-east-1";

   Aws::Lambda::LambdaClient client(clientConfig);

   std::vector<Aws::String> functions;
   Aws::String marker;

   do {
       Aws::Lambda::Model::ListFunctionsRequest request;
       if (!marker.empty()) {
           request.SetMarker(marker);
       }

       Aws::Lambda::Model::ListFunctionsOutcome outcome = client.ListFunctions(
               request);

       if (outcome.IsSuccess()) {
           const Aws::Lambda::Model::ListFunctionsResult &result =
outcome.GetResult();
           std::cout << result.GetFunctions().size()
                     << " lambda functions were retrieved." << std::endl;

           for (const Aws::Lambda::Model::FunctionConfiguration
&functionConfiguration: result.GetFunctions()) {
               functions.push_back(functionConfiguration.GetFunctionName());
               std::cout << functions.size() << "  "
                         << functionConfiguration.GetDescription() << std::endl;
               std::cout << "   "
```

```
                            <<
 Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(
                                    functionConfiguration.GetRuntime()) << ": "
                            << functionConfiguration.GetHandler()
                            << std::endl;
            }
            marker = result.GetNextMarker();
        }
        else {
            std::cerr << "Error with Lambda::ListFunctions. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
    } while (!marker.empty());
```

- Pour plus de détails sur l'API, reportez-vous ListFunctionsà la section Référence des AWS SDK pour C++ API.

CLI

AWS CLI

Pour récupérer la liste des fonctions Lambda

L'exemple `list-functions` suivant affiche une liste de toutes les fonctions pour l'utilisateur actuel.

```
aws lambda list-functions
```

Sortie :

```
{
    "Functions": [
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "dBG9m8SGdmlEjw/JYXlhhvCrAv5TxvXsbL/RMr0fT/I=",
            "FunctionName": "helloworld",
            "MemorySize": 128,
```

```
            "RevisionId": "1718e831-badf-4253-9518-d0644210af7b",
            "CodeSize": 294,
            "FunctionArn": "arn:aws:lambda:us-
west-2:123456789012:function:helloworld",
            "Handler": "helloworld.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/MyTestFunction-
role-zgur6bf4",
            "Timeout": 3,
            "LastModified": "2025-09-23T18:32:33.857+0000",
            "Runtime": "nodejs22.x",
            "Description": ""
        },
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "sU0cJ2/hOZevwV/lTxCuQqK3gDZP3i8gUoqUUVRmY6E=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "93017fc9-59cb-41dc-901b-4845ce4bf668",
            "CodeSize": 266,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2025-10-01T16:47:28.490+0000",
            "Runtime": "nodejs22.x",
            "Description": ""
        },
        {
            "Layers": [
                {
                    "CodeSize": 41784542,
                    "Arn": "arn:aws:lambda:us-
west-2:420165488524:layer:AWSLambda-Python37-SciPy1x:2"
                },
```

```
                {
                    "CodeSize": 4121,
                    "Arn": "arn:aws:lambda:us-
west-2:123456789012:layer:pythonLayer:1"
                }
            ],
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "ZQukCqxtkqFgyF2cU41Avj99TKQ/hNihPtDtRcc08mI=",
            "FunctionName": "my-python-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 128,
            "RevisionId": "80b4eabc-acf7-4ea8-919a-e874c213707d",
            "CodeSize": 299,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
python-function",
            "Handler": "lambda_function.lambda_handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/my-python-
function-role-z5g7dr6n",
            "Timeout": 3,
            "LastModified": "2025-10-01T19:40:41.643+0000",
            "Runtime": "python3.11",
            "Description": ""
        }
    ]
}
```

Pour plus d'informations, consultez Configurer la mémoire de fonction Lambda dans le Guide du développeur AWS Lambda.

- Pour plus de détails sur l'API, voir ListFunctionsla section Référence des AWS CLI commandes.

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// ListFunctions lists up to maxItems functions for the account. This function
 uses a
// lambda.ListFunctionsPaginator to paginate the results.
func (wrapper FunctionWrapper) ListFunctions(ctx context.Context, maxItems int)
 []types.FunctionConfiguration {
 var functions []types.FunctionConfiguration
 paginator := lambda.NewListFunctionsPaginator(wrapper.LambdaClient,
 &lambda.ListFunctionsInput{
  MaxItems: aws.Int32(int32(maxItems)),
 })
```

```
 for paginator.HasMorePages() && len(functions) < maxItems {
  pageOutput, err := paginator.NextPage(ctx)
  if err != nil {
   log.Panicf("Couldn't list functions for your account. Here's why: %v\n", err)
  }
  functions = append(functions, pageOutput.Functions...)
 }
 return functions
}
```

- Pour plus de détails sur l'API, reportez-vous ListFunctionsà la section Référence des AWS SDK pour Go API.

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
const listFunctions = () => {
  const client = new LambdaClient({});
  const command = new ListFunctionsCommand({});

  return client.send(command);
};
```

- Pour plus de détails sur l'API, reportez-vous ListFunctionsà la section Référence des AWS SDK pour JavaScript API.

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    public function listFunctions($maxItems = 50, $marker = null)
    {
        if (is_null($marker)) {
            return $this->lambdaClient->listFunctions([
                'MaxItems' => $maxItems,
            ]);
        }

        return $this->lambdaClient->listFunctions([
            'Marker' => $marker,
            'MaxItems' => $maxItems,
        ]);
    }
```

- Pour plus de détails sur l'API, reportez-vous ListFunctionsà la section Référence des AWS SDK pour PHP API.

PowerShell

Outils pour PowerShell V4

Exemple 1 : cet exemple affiche toutes les fonctions Lambda avec une taille de code triée

```
Get-LMFunctionList | Sort-Object -Property CodeSize | Select-Object FunctionName,
 RunTime, Timeout, CodeSize
```

Sortie :

```
FunctionName                                                Runtime   Timeout
 CodeSize
```

```
------------                                                    -------   -------
 --------
test                                                            python2.7       3
   243
MylambdaFunction123                                             python3.8     600
   659
myfuncpython1                                                   python3.8     303
   675
```

- Pour plus de détails sur l'API, reportez-vous ListFunctionsà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : cet exemple affiche toutes les fonctions Lambda avec une taille de code triée

```
Get-LMFunctionList | Sort-Object -Property CodeSize | Select-Object FunctionName,
 RunTime, Timeout, CodeSize
```

Sortie :

```
FunctionName                                                    Runtime   Timeout
 CodeSize
------------                                                    -------   -------
 --------
test                                                            python2.7       3
   243
MylambdaFunction123                                             python3.8     600
   659
myfuncpython1                                                   python3.8     303
   675
```

- Pour plus de détails sur l'API, reportez-vous ListFunctionsà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def list_functions(self):
        """
        Lists the Lambda functions for the current account.
        """
        try:
            func_paginator = self.lambda_client.get_paginator("list_functions")
            for func_page in func_paginator.paginate():
                for func in func_page["Functions"]:
                    print(func["FunctionName"])
                    desc = func.get("Description")
                    if desc:
                        print(f"\t{desc}")
                    print(f"\t{func['Runtime']}: {func['Handler']}")
        except ClientError as err:
            logger.error(
                "Couldn't list functions. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

- Pour plus de détails sur l'API, consultez ListFunctionsle AWS manuel de référence de l'API SDK for Python (Boto3).

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Lists the Lambda functions for the current account.
  def list_functions
    functions = []
    @lambda_client.list_functions.each do |response|
      response['functions'].each do |function|
        functions.append(function['function_name'])
      end
    end
    functions
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error listing functions:\n #{e.message}")
  end
```

- Pour plus de détails sur l'API, reportez-vous ListFunctionsà la section Référence des AWS SDK pour Ruby API.

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    /** List all Lambda functions in the current Region. */
    pub async fn list_functions(&self) -> Result<ListFunctionsOutput,
anyhow::Error> {
        info!("Listing lambda functions");
        self.lambda_client
            .list_functions()
            .send()
            .await
            .map_err(anyhow::Error::from)
    }
```

- Pour plus de détails sur l'API, voir ListFunctionsla section de référence de l'API AWS SDK for Rust.

SAP ABAP

Kit SDK pour SAP ABAP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    TRY.
        oo_result = lo_lmd->listfunctions( ).       " oo_result is returned for
testing purposes. "
        DATA(lt_functions) = oo_result->get_functions( ).
        MESSAGE 'Retrieved list of Lambda functions.' TYPE 'I'.
```

```
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
 TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- Pour plus de détails sur l'API, reportez-vous ListFunctionsà la section de référence du AWS SDK pour l'API SAP ABAP.

Swift

Kit SDK pour Swift

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
import AWSClientRuntime
import AWSLambda
import Foundation

    /// Returns an array containing the names of all AWS Lambda functions
    /// available to the user.
    ///
    /// - Parameter lambdaClient: The `IAMClient` to use.
    ///
    /// - Throws: `ExampleError.listFunctionsError`
    ///
    /// - Returns: An array of lambda function name strings.
    func getFunctionNames(lambdaClient: LambdaClient) async throws -> [String] {
        let pages = lambdaClient.listFunctionsPaginated(
            input: ListFunctionsInput()
        )

        var functionNames: [String] = []
```

```
        for try await page in pages {
            guard let functions = page.functions else {
                throw ExampleError.listFunctionsError
            }

            for function in functions {
                functionNames.append(function.functionName ?? "<unknown>")
            }
        }

        return functionNames
    }
```

- Pour plus de détails sur l'API, reportez-vous ListFunctionsà la section AWS SDK pour la référence de l'API Swift.

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation de **`ListProvisionedConcurrencyConfigs`** avec une CLI

Les exemples de code suivants illustrent comment utiliser `ListProvisionedConcurrencyConfigs`.

CLI

AWS CLI

Pour obtenir une liste des configurations de simultanéité allouée

L'exemple `list-provisioned-concurrency-configs` suivant répertorie les configurations de simultanéité allouée pour la fonction spécifiée.

```
aws lambda list-provisioned-concurrency-configs \
    --function-name my-function
```

Sortie :

```
{
```

```
    "ProvisionedConcurrencyConfigs": [
        {
            "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-
function:GREEN",
            "RequestedProvisionedConcurrentExecutions": 100,
            "AvailableProvisionedConcurrentExecutions": 100,
            "AllocatedProvisionedConcurrentExecutions": 100,
            "Status": "READY",
            "LastModified": "2019-12-31T20:29:00+0000"
        },
        {
            "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-
function:BLUE",
            "RequestedProvisionedConcurrentExecutions": 100,
            "AvailableProvisionedConcurrentExecutions": 100,
            "AllocatedProvisionedConcurrentExecutions": 100,
            "Status": "READY",
            "LastModified": "2019-12-31T20:28:49+0000"
        }
    ]
}
```

- Pour plus de détails sur l'API, voir ListProvisionedConcurrencyConfigsla section Référence des AWS CLI commandes.

PowerShell

Outils pour PowerShell V4

Exemple 1 : cet exemple récupère la liste des configurations de la simultanéité allouée pour une fonction Lambda.

```
Get-LMProvisionedConcurrencyConfigList -FunctionName "MylambdaFunction123"
```

- Pour plus de détails sur l'API, reportez-vous ListProvisionedConcurrencyConfigsà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : cet exemple récupère la liste des configurations de la simultanéité allouée pour une fonction Lambda.

```
Get-LMProvisionedConcurrencyConfigList -FunctionName "MylambdaFunction123"
```

- Pour plus de détails sur l'API, reportez-vous ListProvisionedConcurrencyConfigsà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation de **`ListTags`** avec une CLI

Les exemples de code suivants illustrent comment utiliser `ListTags`.

CLI

AWS CLI

Pour récupérer la liste des balises d'une fonction Lambda

L'exemple `list-tags` suivant affiche les balises associées à la fonction Lambda `my-function`.

```
aws lambda list-tags \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function
```

Sortie :

```
{
    "Tags": {
        "Category": "Web Tools",
        "Department": "Sales"
    }
}
```

Pour plus d'informations, consultez Étiquetage des fonctions Lambda dans le Guide du développeur AWS .

- Pour plus de détails sur l'API, voir ListTagsla section Référence des AWS CLI commandes.

PowerShell

Outils pour PowerShell V4

Exemple 1 : récupère les balises et leurs valeurs actuellement définies sur la fonction spécifiée.

```
Get-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction"
```

Sortie :

```
Key        Value
---        -----
California Sacramento
Oregon     Salem
Washington Olympia
```

- Pour plus de détails sur l'API, reportez-vous ListTagsà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : récupère les balises et leurs valeurs actuellement définies sur la fonction spécifiée.

```
Get-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction"
```

Sortie :

```
Key        Value
---        -----
California Sacramento
Oregon     Salem
Washington Olympia
```

- Pour plus de détails sur l'API, reportez-vous ListTagsà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation de **`ListVersionsByFunction`** avec une CLI

Les exemples de code suivants illustrent comment utiliser `ListVersionsByFunction`.

CLI

AWS CLI

Pour récupérer la liste des versions d'une fonction

L'exemple `list-versions-by-function` suivant affiche la liste des versions de la fonction Lambda `my-function`.

```
aws lambda list-versions-by-function \
    --function-name my-function
```

Sortie :

```
{
    "Versions": [
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "sU0cJ2/hOZevwV/lTxCuQqK3gDZP3i8gUoqUUVRmY6E=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "93017fc9-59cb-41dc-901b-4845ce4bf668",
            "CodeSize": 266,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:$LATEST",
            "Handler": "index.handler",
```

```
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2019-10-01T16:47:28.490+0000",
            "Runtime": "nodejs10.x",
            "Description": ""
        },
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "1",
            "CodeSha256": "5tT2qgzYUHoqwR616pZ2dpkn/0J1FrzJmlKidWaaCgk=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "949c8914-012e-4795-998c-e467121951b1",
            "CodeSize": 304,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:1",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2019-09-26T20:28:40.438+0000",
            "Runtime": "nodejs10.x",
            "Description": "new version"
        },
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "2",
            "CodeSha256": "sU0cJ2/hOZevwV/lTxCuQqK3gDZP3i8gUoqUUVRmY6E=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
```

```
            "MemorySize": 256,
            "RevisionId": "cd669f21-0f3d-4e1c-9566-948837f2e2ea",
            "CodeSize": 266,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:2",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2019-10-01T16:47:28.490+0000",
            "Runtime": "nodejs10.x",
            "Description": "newer version"
        }
    ]
}
```

Pour plus d'informations, consultez la section Configuration des alias de fonction AWS Lambda dans le guide du développeur Lambda AWS .

- Pour plus de détails sur l'API, voir ListVersionsByFunctionla section Référence des AWS CLI commandes.

PowerShell

Outils pour PowerShell V4

Exemple 1 : cet exemple renvoie la liste des configurations spécifiques à chaque version de la fonction Lambda.

```
Get-LMVersionsByFunction -FunctionName "MylambdaFunction123"
```

Sortie :

```
FunctionName        Runtime   MemorySize Timeout CodeSize LastModified
      RoleName
------------        -------   ---------- ------- -------- ------------
      --------
MylambdaFunction123 python3.8        128     600      659
 2020-01-10T03:20:56.390+0000 lambda
MylambdaFunction123 python3.8        128       5     1426
 2019-12-25T09:19:02.238+0000 lambda
```

```
MylambdaFunction123 python3.8        128       5     1426
 2019-12-25T09:39:36.779+0000 lambda
MylambdaFunction123 python3.8        128     600     1426
 2019-12-25T09:52:59.872+0000 lambda
```

- Pour plus de détails sur l'API, reportez-vous ListVersionsByFunctionà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : cet exemple renvoie la liste des configurations spécifiques à chaque version de la fonction Lambda.

```
Get-LMVersionsByFunction -FunctionName "MylambdaFunction123"
```

Sortie :

```
FunctionName        Runtime   MemorySize Timeout CodeSize LastModified
      RoleName
------------        -------   ---------- ------- -------- ------------
      --------
MylambdaFunction123 python3.8        128     600      659
 2020-01-10T03:20:56.390+0000 lambda
MylambdaFunction123 python3.8        128       5     1426
 2019-12-25T09:19:02.238+0000 lambda
MylambdaFunction123 python3.8        128       5     1426
 2019-12-25T09:39:36.779+0000 lambda
MylambdaFunction123 python3.8        128     600     1426
 2019-12-25T09:52:59.872+0000 lambda
```

- Pour plus de détails sur l'API, reportez-vous ListVersionsByFunctionà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation de **`PublishVersion`** avec une CLI

Les exemples de code suivants illustrent comment utiliser `PublishVersion`.

CLI

AWS CLI

Pour publier une nouvelle version d'une fonction

L'exemple `publish-version` suivant publie une nouvelle version de la fonction Lambda `my-function`.

```
aws lambda publish-version \
    --function-name my-function
```

Sortie :

```
{
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "dBG9m8SGdmlEjw/JYXlhhvCrAv5TxvXsbL/RMr0fT/I=",
    "FunctionName": "my-function",
    "CodeSize": 294,
    "RevisionId": "f31d3d39-cc63-4520-97d4-43cd44c94c20",
    "MemorySize": 128,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:3",
    "Version": "2",
    "Role": "arn:aws:iam::123456789012:role/service-role/MyTestFunction-role-
zgur6bf4",
    "Timeout": 3,
    "LastModified": "2019-09-23T18:32:33.857+0000",
    "Handler": "my-function.handler",
    "Runtime": "nodejs10.x",
    "Description": ""
}
```

Pour plus d'informations, consultez la section Configuration des alias de fonction AWS Lambda dans le guide du développeur Lambda AWS .

- Pour plus de détails sur l'API, voir PublishVersionla section Référence des AWS CLI commandes.

PowerShell

Outils pour PowerShell V4

Exemple 1 : cet exemple crée une version pour l'instantané existant du code de fonction Lambda

```
Publish-LMVersion -FunctionName "MylambdaFunction123" -Description "Publishing
 Existing Snapshot of function code as a  new version through Powershell"
```

- Pour plus de détails sur l'API, reportez-vous PublishVersionà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : cet exemple crée une version pour l'instantané existant du code de fonction Lambda

```
Publish-LMVersion -FunctionName "MylambdaFunction123" -Description "Publishing
 Existing Snapshot of function code as a  new version through Powershell"
```

- Pour plus de détails sur l'API, reportez-vous PublishVersionà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation de **`PutFunctionConcurrency`** avec une CLI

Les exemples de code suivants illustrent comment utiliser `PutFunctionConcurrency`.

CLI

AWS CLI

Pour configurer une limite de simultanéité réservée pour une fonction

L'exemple `put-function-concurrency` suivant configure 100 exécutions simultanées réservées pour la fonction `my-function`.

```
aws lambda put-function-concurrency \
    --function-name  my-function  \
    --reserved-concurrent-executions 100
```

Sortie :

```
{
    "ReservedConcurrentExecutions": 100
}
```

Pour plus d'informations, consultez Réservation de simultanéité pour une fonction Lambda dans le Guide du développeur AWS Lambda.

- Pour plus de détails sur l'API, voir PutFunctionConcurrencyla section Référence des AWS CLI commandes.

PowerShell

Outils pour PowerShell V4

Exemple 1 : cet exemple applique les paramètres de simultanéité pour la fonction dans son ensemble.

```
Write-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -
ReservedConcurrentExecution 100
```

- Pour plus de détails sur l'API, reportez-vous PutFunctionConcurrencyà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : cet exemple applique les paramètres de simultanéité pour la fonction dans son ensemble.

```
Write-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -
ReservedConcurrentExecution 100
```

- Pour plus de détails sur l'API, reportez-vous PutFunctionConcurrencyà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation de **PutProvisionedConcurrencyConfig** avec une CLI

Les exemples de code suivants illustrent comment utiliser `PutProvisionedConcurrencyConfig`.

CLI

AWS CLI

Pour allouer une simultanéité allouée

L'exemple `put-provisioned-concurrency-config` suivant alloue 100 simultanéités allouées à l'alias BLUE de la fonction spécifiée.

```
aws lambda put-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier BLUE \
    --provisioned-concurrent-executions 100
```

Sortie :

```
{
    "Requested ProvisionedConcurrentExecutions": 100,
    "Allocated ProvisionedConcurrentExecutions": 0,
    "Status": "IN_PROGRESS",
    "LastModified": "2019-11-21T19:32:12+0000"
}
```

- Pour plus de détails sur l'API, voir PutProvisionedConcurrencyConfigla section Référence des AWS CLI commandes.

PowerShell

Outils pour PowerShell V4

Exemple 1 : cet exemple ajoute une configuration de simultanéité allouée à l'alias d'une fonction

```
Write-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
ProvisionedConcurrentExecution 20 -Qualifier "NewAlias1"
```

- Pour plus de détails sur l'API, reportez-vous PutProvisionedConcurrencyConfigà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : cet exemple ajoute une configuration de simultanéité allouée à l'alias d'une fonction

```
Write-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
ProvisionedConcurrentExecution 20 -Qualifier "NewAlias1"
```

- Pour plus de détails sur l'API, reportez-vous PutProvisionedConcurrencyConfigà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation de **`RemovePermission`** avec une CLI

Les exemples de code suivants illustrent comment utiliser `RemovePermission`.

CLI

AWS CLI

Pour supprimer les autorisations d'une fonction Lambda existante

L'exemple `remove-permission` retire l'autorisation d'invoquer une fonction nommée `my-function`.

```
aws lambda remove-permission \
    --function-name my-function \
    --statement-id sns
```

Cette commande ne produit aucune sortie.

Pour plus d'informations, consultez la section Utilisation de politiques basées sur les ressources pour Lambda AWS dans le guide du développeur AWS Lambda.

- Pour plus de détails sur l'API, voir RemovePermissionla section Référence des AWS CLI commandes.

PowerShell

Outils pour PowerShell V4

Exemple 1 : Cet exemple supprime la politique de fonction pour la fonction Lambda spécifiée StatementId .

```
$policy =  Get-LMPolicy -FunctionName "MylambdaFunction123" -Select Policy |
 ConvertFrom-Json| Select-Object -ExpandProperty Statement
Remove-LMPermission -FunctionName "MylambdaFunction123" -StatementId
 $policy[0].Sid
```

- Pour plus de détails sur l'API, reportez-vous RemovePermissionà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : Cet exemple supprime la politique de fonction pour la fonction Lambda spécifiée StatementId .

```
$policy =  Get-LMPolicy -FunctionName "MylambdaFunction123" -Select Policy |
 ConvertFrom-Json| Select-Object -ExpandProperty Statement
Remove-LMPermission -FunctionName "MylambdaFunction123" -StatementId
 $policy[0].Sid
```

- Pour plus de détails sur l'API, reportez-vous RemovePermissionà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation de **`TagResource`** avec une CLI

Les exemples de code suivants illustrent comment utiliser `TagResource`.

CLI

AWS CLI

Pour ajouter des balises à une fonction Lambda existante

L'exemple `tag-resource` suivant ajoute une balise avec le nom de clé `DEPARTMENT` et la valeur `Department A` à la fonction Lambda spécifiée.

```
aws lambda tag-resource \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function \
    --tags "DEPARTMENT=Department A"
```

Cette commande ne produit aucune sortie.

Pour plus d'informations, consultez Étiquetage des fonctions Lambda dans le Guide du développeur AWS .

- Pour plus de détails sur l'API, voir TagResourcela section Référence des AWS CLI commandes.

PowerShell

Outils pour PowerShell V4

Exemple 1 : ajoute les trois balises (Washington, Oregon et Californie) et leurs valeurs associées à la fonction spécifiée identifiée par son ARN.

```
Add-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -Tag @{ "Washington" = "Olympia";
 "Oregon" = "Salem"; "California" = "Sacramento" }
```

- Pour plus de détails sur l'API, reportez-vous TagResourceà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : ajoute les trois balises (Washington, Oregon et Californie) et leurs valeurs associées à la fonction spécifiée identifiée par son ARN.

```
Add-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -Tag @{ "Washington" = "Olympia";
 "Oregon" = "Salem"; "California" = "Sacramento" }
```

- Pour plus de détails sur l'API, reportez-vous TagResourceà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation de **`UntagResource`** avec une CLI

Les exemples de code suivants illustrent comment utiliser `UntagResource`.

CLI

AWS CLI

Pour supprimer les balises d'une fonction Lambda existante

L'exemple `untag-resource` suivant supprime la balise dont la clé est `DEPARTMENT` de la fonction Lambda `my-function`.

```
aws lambda untag-resource \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function \
    --tag-keys DEPARTMENT
```

Cette commande ne produit aucune sortie.

Pour plus d'informations, consultez Étiquetage des fonctions Lambda dans le Guide du développeur AWS .

- Pour plus de détails sur l'API, voir UntagResourcela section Référence des AWS CLI commandes.

PowerShell

Outils pour PowerShell V4

Exemple 1 : supprime les balises fournies d'une fonction. L'applet de commande vous invitera à confirmer avant de poursuivre, à moins que l'option -Force ne soit spécifiée. Un seul appel est envoyé au service pour supprimer les tags.

```
Remove-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -TagKey
 "Washington","Oregon","California"
```

Exemple 2 : supprime les balises fournies d'une fonction. L'applet de commande vous invitera à confirmer avant de poursuivre, à moins que l'option -Force ne soit spécifiée. Un seul appel au service est effectué par tag fourni.

```
"Washington","Oregon","California" | Remove-LMResourceTag -Resource
 "arn:aws:lambda:us-west-2:123456789012:function:MyFunction"
```

- Pour plus de détails sur l'API, reportez-vous UntagResourceà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : supprime les balises fournies d'une fonction. L'applet de commande vous invitera à confirmer avant de poursuivre, à moins que l'option -Force ne soit spécifiée. Un seul appel est envoyé au service pour supprimer les tags.

```
Remove-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -TagKey
 "Washington","Oregon","California"
```

Exemple 2 : supprime les balises fournies d'une fonction. L'applet de commande vous invitera à confirmer avant de poursuivre, à moins que l'option -Force ne soit spécifiée. Un seul appel au service est effectué par tag fourni.

```
"Washington","Oregon","California" | Remove-LMResourceTag -Resource
 "arn:aws:lambda:us-west-2:123456789012:function:MyFunction"
```

- Pour plus de détails sur l'API, reportez-vous UntagResourceà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation de **`UpdateAlias`** avec une CLI

Les exemples de code suivants illustrent comment utiliser `UpdateAlias`.

CLI

AWS CLI

Pour mettre à jour un alias de fonction

L'exemple `update-alias` suivant met à jour l'alias de fonction Lambda nommé `LIVE` pour qu'il pointe vers la version 3 de la fonction Lambda `my-function`.

```
aws lambda update-alias \
    --function-name my-function \
    --function-version 3 \
    --name LIVE
```

Sortie :

```
{
    "FunctionVersion": "3",
    "Name": "LIVE",
    "AliasArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:LIVE",
    "RevisionId": "594f41fb-b85f-4c20-95c7-6ca5f2a92c93",
    "Description": "alias for live version of function"
}
```

Pour plus d'informations, consultez la section Configuration des alias de fonction AWS Lambda dans le guide du développeur Lambda AWS .

- Pour plus de détails sur l'API, voir UpdateAliasla section Référence des AWS CLI commandes.

PowerShell

Outils pour PowerShell V4

Exemple 1 : cet exemple met à jour la configuration d'un alias de fonction Lambda existant. Il met à jour la RoutingConfiguration valeur pour transférer 60 % (0,6) du trafic vers la version 1

```
Update-LMAlias -FunctionName "MylambdaFunction123" -Description
 " Alias for version 2" -FunctionVersion 2 -Name "newlabel1" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6}
```

- Pour plus de détails sur l'API, reportez-vous UpdateAliasà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : cet exemple met à jour la configuration d'un alias de fonction Lambda existant. Il met à jour la RoutingConfiguration valeur pour transférer 60 % (0,6) du trafic vers la version 1

```
Update-LMAlias -FunctionName "MylambdaFunction123" -Description
 " Alias for version 2" -FunctionVersion 2 -Name "newlabel1" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6}
```

- Pour plus de détails sur l'API, reportez-vous UpdateAliasà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation **`UpdateFunctionCode`** avec un AWS SDK ou une CLI

Les exemples de code suivants illustrent comment utiliser `UpdateFunctionCode`.

Les exemples d'actions sont des extraits de code de programmes de plus grande envergure et doivent être exécutés en contexte. Vous pouvez voir cette action en contexte dans l'exemple de code suivant :

- Principes de base

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    /// <summary>
    /// Update an existing Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to update.</
param>
    /// <param name="bucketName">The bucket where the zip file containing
    /// the Lambda function code is stored.</param>
    /// <param name="key">The key name of the source code file.</param>
    /// <returns>Async Task.</returns>
    public async Task UpdateFunctionCodeAsync(
        string functionName,
        string bucketName,
        string key)
    {
        var functionCodeRequest = new UpdateFunctionCodeRequest
        {
            FunctionName = functionName,
            Publish = true,
            S3Bucket = bucketName,
            S3Key = key,
        };

        var response = await
 _lambdaService.UpdateFunctionCodeAsync(functionCodeRequest);
        Console.WriteLine($"The Function was last modified at
 {response.LastModified}.");
    }
```

- Pour plus de détails sur l'API, reportez-vous UpdateFunctionCodeà la section Référence des AWS SDK pour .NET API.

C++

SDK pour C++

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
 (overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::UpdateFunctionCodeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        std::ifstream ifstream(CALCULATOR_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteLambdaFunction(client);
            deleteIamRole(clientConfig);
            return false;
        }

        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();
        request.SetZipFile(
                Aws::Utils::ByteBuffer((unsigned char *) buffer.str().c_str(),
                                       buffer.str().length()));
        request.SetPublish(true);
```

```
        Aws::Lambda::Model::UpdateFunctionCodeOutcome outcome =
 client.UpdateFunctionCode(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda code was successfully updated." <<
 std::endl;
        }
        else {
            std::cerr << "Error with Lambda::UpdateFunctionCode. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
```

- Pour plus de détails sur l'API, reportez-vous UpdateFunctionCodeà la section Référence des AWS SDK pour C++ API.

CLI

AWS CLI

Pour mettre à jour le code d'une fonction Lambda

L'exemple `update-function-code` suivant remplace le code de la version non publiée ($LATEST) de la fonction `my-function` par le contenu du fichier zip spécifié.

```
aws lambda update-function-code \
    --function-name  my-function \
    --zip-file fileb://my-function.zip
```

Sortie :

```
{
    "FunctionName": "my-function",
    "LastModified": "2019-09-26T20:28:40.438+0000",
    "RevisionId": "e52502d4-9320-4688-9cd6-152a6ab7490d",
    "MemorySize": 256,
    "Version": "$LATEST",
    "Role": "arn:aws:iam::123456789012:role/service-role/my-function-role-
uy3l9qyq",
```

```
    "Timeout": 3,
    "Runtime": "nodejs10.x",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "5tT2qgzYUHaqwR716pZ2dpkn/0J1FrzJmlKidWoaCgk=",
    "Description": "",
    "VpcConfig": {
        "SubnetIds": [],
        "VpcId": "",
        "SecurityGroupIds": []
    },
    "CodeSize": 304,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "Handler": "index.handler"
}
```

Pour plus d'informations, consultez Configuration des options de fonction Lambda AWS dans le Guide du développeur AWS .

- Pour plus de détails sur l'API, voir UpdateFunctionCodela section Référence des AWS CLI commandes.

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"
```

```
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// UpdateFunctionCode updates the code for the Lambda function specified by
 functionName.
// The existing code for the Lambda function is entirely replaced by the code in
 the
// zipPackage buffer. After the update action is called, a
 lambda.FunctionUpdatedV2Waiter
// is used to wait until the update is successful.
func (wrapper FunctionWrapper) UpdateFunctionCode(ctx context.Context,
 functionName string, zipPackage *bytes.Buffer) types.State {
 var state types.State
 _, err := wrapper.LambdaClient.UpdateFunctionCode(ctx,
 &lambda.UpdateFunctionCodeInput{
  FunctionName: aws.String(functionName), ZipFile: zipPackage.Bytes(),
 })
 if err != nil {
  log.Panicf("Couldn't update code for function %v. Here's why: %v\n",
 functionName, err)
 } else {
  waiter := lambda.NewFunctionUpdatedV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(ctx, &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
 functionName, err)
  } else {
   state = funcOutput.Configuration.State
  }
 }
 return state
}
```

- Pour plus de détails sur l'API, reportez-vous UpdateFunctionCodeà la section Référence des AWS SDK pour Go API.

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
   /**
    * Updates the code for an AWS Lambda function.
    *
    * @param awsLambda  the AWS Lambda client
    * @param functionName the name of the Lambda function to update
    * @param bucketName the name of the S3 bucket where the function code is
located
    * @param key the key (file name) of the function code in the S3 bucket
    * @throws LambdaException if there is an error updating the function code
    */
   public static void updateFunctionCode(LambdaClient awsLambda, String
functionName, String bucketName, String key) {
       try {
           LambdaWaiter waiter = awsLambda.waiter();
           UpdateFunctionCodeRequest functionCodeRequest =
UpdateFunctionCodeRequest.builder()
                .functionName(functionName)
                .publish(true)
                .s3Bucket(bucketName)
                .s3Key(key)
                .build();

           UpdateFunctionCodeResponse response =
awsLambda.updateFunctionCode(functionCodeRequest);
           GetFunctionConfigurationRequest getFunctionConfigRequest =
GetFunctionConfigurationRequest.builder()
```

```
                .functionName(functionName)
                .build();

            WaiterResponse<GetFunctionConfigurationResponse> waiterResponse =
 waiter
                .waitUntilFunctionUpdated(getFunctionConfigRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            System.out.println("The last modified value is " +
 response.lastModified());

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

- Pour plus de détails sur l'API, reportez-vous UpdateFunctionCodeà la section Référence des AWS SDK for Java 2.x API.

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
const updateFunctionCode = async (funcName, newFunc) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${newFunc}.zip`);
  const command = new UpdateFunctionCodeCommand({
    ZipFile: code,
    FunctionName: funcName,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });
```

```
  return client.send(command);
};
```

- Pour plus de détails sur l'API, reportez-vous UpdateFunctionCodeà la section Référence des AWS SDK pour JavaScript API.

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    public function updateFunctionCode($functionName, $s3Bucket, $s3Key)
    {
        return $this->lambdaClient->updateFunctionCode([
            'FunctionName' => $functionName,
            'S3Bucket' => $s3Bucket,
            'S3Key' => $s3Key,
        ]);
    }
```

- Pour plus de détails sur l'API, reportez-vous UpdateFunctionCodeà la section Référence des AWS SDK pour PHP API.

PowerShell

Outils pour PowerShell V4

Exemple 1 : met à jour la fonction nommée MyFunction « » avec le nouveau contenu contenu dans le fichier zip spécifié. Pour une fonction Lambda C# .NET Core, le fichier zip doit contenir l'assemblage compilé.

```
Update-LMFunctionCode -FunctionName MyFunction -ZipFilename .\UpdatedCode.zip
```

Exemple 2 : cet exemple est similaire au précédent, mais utilise un objet Amazon S3 contenant le code mis à jour pour actualiser la fonction.

```
Update-LMFunctionCode -FunctionName MyFunction -BucketName amzn-s3-demo-bucket -
Key UpdatedCode.zip
```

- Pour plus de détails sur l'API, reportez-vous UpdateFunctionCodeà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : met à jour la fonction nommée MyFunction « » avec le nouveau contenu contenu dans le fichier zip spécifié. Pour une fonction Lambda C# .NET Core, le fichier zip doit contenir l'assemblage compilé.

```
Update-LMFunctionCode -FunctionName MyFunction -ZipFilename .\UpdatedCode.zip
```

Exemple 2 : cet exemple est similaire au précédent, mais utilise un objet Amazon S3 contenant le code mis à jour pour actualiser la fonction.

```
Update-LMFunctionCode -FunctionName MyFunction -BucketName amzn-s3-demo-bucket -
Key UpdatedCode.zip
```

- Pour plus de détails sur l'API, reportez-vous UpdateFunctionCodeà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
class LambdaWrapper:
```

```
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def update_function_code(self, function_name, deployment_package):
        """
        Updates the code for a Lambda function by submitting a .zip archive that
contains
        the code for the function.

        :param function_name: The name of the function to update.
        :param deployment_package: The function code to update, packaged as bytes
in
                                   .zip format.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_code(
                FunctionName=function_name, ZipFile=deployment_package
            )
        except ClientError as err:
            logger.error(
                "Couldn't update function %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response
```

- Pour plus de détails sur l'API, consultez UpdateFunctionCodele AWS manuel de référence de l'API SDK for Python (Boto3).

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Updates the code for a Lambda function by submitting a .zip archive that
 contains
  # the code for the function.
  #
  # @param function_name: The name of the function to update.
  # @param deployment_package: The function code to update, packaged as bytes in
  #                            .zip format.
  # @return: Data about the update, including the status.
  def update_function_code(function_name, deployment_package)
    @lambda_client.update_function_code(
      function_name: function_name,
      zip_file: deployment_package
    )
    @lambda_client.wait_until(:function_updated_v2, { function_name:
 function_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error updating function code for:
 #{function_name}:\n #{e.message}")
```

```
    nil
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to update:\n
#{e.message}")
  end
```

- Pour plus de détails sur l'API, reportez-vous UpdateFunctionCodeà la section Référence des AWS SDK pour Ruby API.

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    /** Given a Path to a zip file, update the function's code and wait for the
update to finish. */
    pub async fn update_function_code(
        &self,
        zip_file: PathBuf,
        key: String,
    ) -> Result<UpdateFunctionCodeOutput, anyhow::Error> {
        let function_code = self.prepare_function(zip_file, Some(key)).await?;

        info!("Updating code for {}", self.lambda_name);
        let update = self
            .lambda_client
            .update_function_code()
            .function_name(self.lambda_name.clone())
            .s3_bucket(self.bucket.clone())
            .s3_key(function_code.s3_key().unwrap().to_string())
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;
```

```
        Ok(update)
    }

    /**
     * Upload function code from a path to a zip file.
     * The zip file must have an AL2 Linux-compatible binary called `bootstrap`.
     * The easiest way to create such a zip is to use `cargo lambda build --
output-format Zip`.
     */
    async fn prepare_function(
        &self,
        zip_file: PathBuf,
        key: Option<String>,
    ) -> Result<FunctionCode, anyhow::Error> {
        let body = ByteStream::from_path(zip_file).await?;

        let key = key.unwrap_or_else(|| format!("{}_code", self.lambda_name));

        info!("Uploading function code to s3://{}/{}", self.bucket, key);
        let _ = self
            .s3_client
            .put_object()
            .bucket(self.bucket.clone())
            .key(key.clone())
            .body(body)
            .send()
            .await?;

        Ok(FunctionCode::builder()
            .s3_bucket(self.bucket.clone())
            .s3_key(key)
            .build())
    }
```

- Pour plus de détails sur l'API, voir UpdateFunctionCodela section de référence de l'API AWS SDK for Rust.

SAP ABAP

Kit SDK pour SAP ABAP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
   TRY.
       oo_result = lo_lmd->updatefunctioncode(     " oo_result is returned for
testing purposes. "
             iv_functionname = iv_function_name
             iv_zipfile = io_zip_file ).

       MESSAGE 'Lambda function code updated.' TYPE 'I'.
     CATCH /aws1/cx_lmdcodesigningcfgno00.
       MESSAGE 'Code signing configuration does not exist.' TYPE 'E'.
     CATCH /aws1/cx_lmdcodestorageexcdex.
       MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
     CATCH /aws1/cx_lmdcodeverification00.
       MESSAGE 'Code signature failed one or more validation checks for
signature mismatch or expiration.' TYPE 'E'.
     CATCH /aws1/cx_lmdinvalidcodesigex.
       MESSAGE 'Code signature failed the integrity check.' TYPE 'E'.
     CATCH /aws1/cx_lmdinvparamvalueex.
       MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
     CATCH /aws1/cx_lmdresourceconflictex.
       MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
     CATCH /aws1/cx_lmdresourcenotfoundex.
       MESSAGE 'The requested resource does not exist.' TYPE 'E'.
     CATCH /aws1/cx_lmdserviceexception.
       MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
     CATCH /aws1/cx_lmdtoomanyrequestsex.
       MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
   ENDTRY.
```

- Pour plus de détails sur l'API, reportez-vous UpdateFunctionCodeà la section de référence du AWS SDK pour l'API SAP ABAP.

Swift

Kit SDK pour Swift

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
import AWSClientRuntime
import AWSLambda
import Foundation

        let zipUrl = URL(fileURLWithPath: path)
        let zipData: Data

        // Read the function's Zip file.

        do {
            zipData = try Data(contentsOf: zipUrl)
        } catch {
            throw ExampleError.zipFileReadError
        }

        // Update the function's code and wait for the updated version to be
        // ready for use.

        do {
            _ = try await lambdaClient.updateFunctionCode(
                input: UpdateFunctionCodeInput(
                    functionName: functionName,
                    zipFile: zipData
                )
            )
        } catch {
            return false
        }
```

- Pour plus de détails sur l'API, reportez-vous UpdateFunctionCodeà la section AWS SDK pour la référence de l'API Swift.

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisation **`UpdateFunctionConfiguration`** avec un AWS SDK ou une CLI

Les exemples de code suivants illustrent comment utiliser `UpdateFunctionConfiguration`.

Les exemples d'actions sont des extraits de code de programmes de plus grande envergure et doivent être exécutés en contexte. Vous pouvez voir cette action en contexte dans l'exemple de code suivant :

- Principes de base

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
   /// <summary>
   /// Update the code of a Lambda function.
   /// </summary>
   /// <param name="functionName">The name of the function to update.</param>
   /// <param name="functionHandler">The code that performs the function's
actions.</param>
   /// <param name="environmentVariables">A dictionary of environment
variables.</param>
   /// <returns>A Boolean value indicating the success of the action.</returns>
   public async Task<bool> UpdateFunctionConfigurationAsync(
       string functionName,
```

```
        string functionHandler,
        Dictionary<string, string> environmentVariables)
    {
        var request = new UpdateFunctionConfigurationRequest
        {
            Handler = functionHandler,
            FunctionName = functionName,
            Environment = new Amazon.Lambda.Model.Environment { Variables =
environmentVariables },
        };

        var response = await
_lambdaService.UpdateFunctionConfigurationAsync(request);

        Console.WriteLine(response.LastModified);

        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }
```

- Pour plus de détails sur l'API, reportez-vous UpdateFunctionConfigurationà la section Référence des AWS SDK pour .NET API.

C++

SDK pour C++

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);
```

```
        Aws::Lambda::Model::UpdateFunctionConfigurationRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        Aws::Lambda::Model::Environment environment;
        environment.AddVariables("LOG_LEVEL", "DEBUG");
        request.SetEnvironment(environment);

        Aws::Lambda::Model::UpdateFunctionConfigurationOutcome outcome =
 client.UpdateFunctionConfiguration(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda configuration was successfully updated."
                      << std::endl;
            break;
        }

        else {
            std::cerr << "Error with Lambda::UpdateFunctionConfiguration. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
```

- Pour plus de détails sur l'API, reportez-vous UpdateFunctionConfigurationà la section Référence des AWS SDK pour C++ API.

CLI

AWS CLI

Pour modifier la configuration d’une fonction

L’exemple `update-function-configuration` suivant modifie la taille de la mémoire à 256 Mo pour la version non publiée ($LATEST) de la fonction `my-function`.

```
aws lambda update-function-configuration \
    --function-name  my-function \
    --memory-size 256
```

Sortie :

```
{
```

```
    "FunctionName": "my-function",
    "LastModified": "2019-09-26T20:28:40.438+0000",
    "RevisionId": "e52502d4-9320-4688-9cd6-152a6ab7490d",
    "MemorySize": 256,
    "Version": "$LATEST",
    "Role": "arn:aws:iam::123456789012:role/service-role/my-function-role-
uy3l9qyq",
    "Timeout": 3,
    "Runtime": "nodejs10.x",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "5tT2qgzYUHaqwR716pZ2dpkn/0J1FrzJmlKidWoaCgk=",
    "Description": "",
    "VpcConfig": {
        "SubnetIds": [],
        "VpcId": "",
        "SecurityGroupIds": []
    },
    "CodeSize": 304,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "Handler": "index.handler"
}
```

Pour plus d'informations, consultez Configuration des options de fonction Lambda AWS dans le Guide du développeur AWS .

- Pour plus de détails sur l'API, voir UpdateFunctionConfigurationla section Référence des AWS CLI commandes.

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// UpdateFunctionConfiguration updates a map of environment variables configured
 for
// the Lambda function specified by functionName.
func (wrapper FunctionWrapper) UpdateFunctionConfiguration(ctx context.Context,
 functionName string, envVars map[string]string) {
 _, err := wrapper.LambdaClient.UpdateFunctionConfiguration(ctx,
 &lambda.UpdateFunctionConfigurationInput{
  FunctionName: aws.String(functionName),
  Environment:  &types.Environment{Variables: envVars},
 })
 if err != nil {
  log.Panicf("Couldn't update configuration for %v. Here's why: %v",
 functionName, err)
 }
}
```

- Pour plus de détails sur l'API, reportez-vous UpdateFunctionConfigurationà la section Référence des AWS SDK pour Go API.

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    /**
     * Updates the configuration of an AWS Lambda function.
     *
     * @param awsLambda     the {@link LambdaClient} instance to use for the AWS
Lambda operation
     * @param functionName  the name of the AWS Lambda function to update
     * @param handler       the new handler for the AWS Lambda function
     *
     * @throws LambdaException if there is an error while updating the function
configuration
     */
    public static void updateFunctionConfiguration(LambdaClient awsLambda, String
functionName, String handler) {
        try {
            UpdateFunctionConfigurationRequest configurationRequest =
UpdateFunctionConfigurationRequest.builder()
                .functionName(functionName)
                .handler(handler)
                .runtime(Runtime.JAVA17)
                .build();

            awsLambda.updateFunctionConfiguration(configurationRequest);

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

- Pour plus de détails sur l'API, reportez-vous UpdateFunctionConfigurationà la section Référence des AWS SDK for Java 2.x API.

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
const updateFunctionConfiguration = (funcName) => {
  const client = new LambdaClient({});
  const config = readFileSync(`${dirname}../functions/config.json`).toString();
  const command = new UpdateFunctionConfigurationCommand({
    ...JSON.parse(config),
    FunctionName: funcName,
  });
  const result = client.send(command);
  waitForFunctionUpdated({ FunctionName: funcName });
  return result;
};
```

- Pour plus de détails sur l'API, reportez-vous UpdateFunctionConfigurationà la section Référence des AWS SDK pour JavaScript API.

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    public function updateFunctionConfiguration($functionName, $handler,
 $environment = '')
    {
```

```
        return $this->lambdaClient->updateFunctionConfiguration([
            'FunctionName' => $functionName,
            'Handler' => "$handler.lambda_handler",
            'Environment' => $environment,
        ]);
    }
```

- Pour plus de détails sur l'API, reportez-vous UpdateFunctionConfigurationà la section Référence des AWS SDK pour PHP API.

PowerShell

Outils pour PowerShell V4

Exemple 1 : cet exemple met à jour la configuration de la fonction Lambda existante

```
Update-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Handler
 "lambda_function.launch_instance" -Timeout 600 -Environment_Variable
 @{ "envvar1"="value";"envvar2"="value" } -Role arn:aws:iam::123456789101:role/
service-role/lambda -DeadLetterConfig_TargetArn arn:aws:sns:us-east-1:
 123456789101:MyfirstTopic
```

- Pour plus de détails sur l'API, reportez-vous UpdateFunctionConfigurationà la section Référence des Outils AWS pour PowerShell applets de commande (V4).

Outils pour PowerShell V5

Exemple 1 : cet exemple met à jour la configuration de la fonction Lambda existante

```
Update-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Handler
 "lambda_function.launch_instance" -Timeout 600 -Environment_Variable
 @{ "envvar1"="value";"envvar2"="value" } -Role arn:aws:iam::123456789101:role/
service-role/lambda -DeadLetterConfig_TargetArn arn:aws:sns:us-east-1:
 123456789101:MyfirstTopic
```

- Pour plus de détails sur l'API, reportez-vous UpdateFunctionConfigurationà la section Référence des Outils AWS pour PowerShell applets de commande (V5).

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def update_function_configuration(self, function_name, env_vars):
        """
        Updates the environment variables for a Lambda function.

        :param function_name: The name of the function to update.
        :param env_vars: A dict of environment variables to update.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_configuration(
                FunctionName=function_name, Environment={"Variables": env_vars}
            )
        except ClientError as err:
            logger.error(
                "Couldn't update function configuration %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response
```

- Pour plus de détails sur l'API, consultez UpdateFunctionConfigurationle AWS manuel de référence de l'API SDK for Python (Boto3).

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Updates the environment variables for a Lambda function.
  # @param function_name: The name of the function to update.
  # @param log_level: The log level of the function.
  # @return: Data about the update, including the status.
  def update_function_configuration(function_name, log_level)
    @lambda_client.update_function_configuration({
                                                   function_name: function_name,
                                                   environment: {
                                                     variables: {
                                                       'LOG_LEVEL' => log_level
                                                     }
                                                   }
                                                 })
    @lambda_client.wait_until(:function_updated_v2, { function_name:
 function_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
```

```
    end
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error updating configurations for
#{function_name}:\n #{e.message}")
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
  end
```

- Pour plus de détails sur l'API, reportez-vous UpdateFunctionConfigurationà la section Référence des AWS SDK pour Ruby API.

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    /** Update the environment for a function. */
    pub async fn update_function_configuration(
        &self,
        environment: Environment,
    ) -> Result<UpdateFunctionConfigurationOutput, anyhow::Error> {
        info!(
            ?environment,
            "Updating environment for {}", self.lambda_name
        );
        let updated = self
            .lambda_client
            .update_function_configuration()
            .function_name(self.lambda_name.clone())
            .environment(environment)
            .send()
            .await
            .map_err(anyhow::Error::from)?;
```

```
        self.wait_for_function_ready().await?;

        Ok(updated)
    }
```

- Pour plus de détails sur l'API, voir UpdateFunctionConfigurationla section de référence de l'API AWS SDK for Rust.

SAP ABAP

Kit SDK pour SAP ABAP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
    TRY.
        oo_result = lo_lmd->updatefunctionconfiguration(     " oo_result is
returned for testing purposes. "
              iv_functionname = iv_function_name
              iv_runtime = iv_runtime
              iv_description  = 'Updated Lambda function'
              iv_memorysize  = iv_memory_size ).

        MESSAGE 'Lambda function configuration/settings updated.' TYPE 'I'.
      CATCH /aws1/cx_lmdcodesigningcfgno00.
        MESSAGE 'Code signing configuration does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdcodeverification00.
        MESSAGE 'Code signature failed one or more validation checks for
signature mismatch or expiration.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvalidcodesigex.
        MESSAGE 'Code signature failed the integrity check.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdresourceconflictex.
        MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
      CATCH /aws1/cx_lmdresourcenotfoundex.
```

```
        MESSAGE 'The requested resource does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
 TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- Pour plus de détails sur l'API, reportez-vous UpdateFunctionConfigurationà la section de référence du AWS SDK pour l'API SAP ABAP.

Swift

Kit SDK pour Swift

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

```
import AWSClientRuntime
import AWSLambda
import Foundation

    /// Tell the server-side component to log debug output by setting its
    /// environment's `LOG_LEVEL` to `DEBUG`.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `LambdaClient` to use.
    ///   - functionName: The name of the AWS Lambda function to enable debug
    ///     logging for.
    ///
    /// - Throws: `ExampleError.environmentResponseMissingError`,
    ///   `ExampleError.updateFunctionConfigurationError`,
    ///   `ExampleError.environmentVariablesMissingError`,
    ///   `ExampleError.logLevelIncorrectError`,
    ///   `ExampleError.updateFunctionConfigurationError`
    func enableDebugLogging(lambdaClient: LambdaClient, functionName: String)
 async throws {
```

```
        let envVariables = [
            "LOG_LEVEL": "DEBUG"
        ]
        let environment = LambdaClientTypes.Environment(variables: envVariables)

        do {
            let output = try await lambdaClient.updateFunctionConfiguration(
                input: UpdateFunctionConfigurationInput(
                    environment: environment,
                    functionName: functionName
                )
            )

            guard let response = output.environment else {
                throw ExampleError.environmentResponseMissingError
            }

            if response.error != nil {
                throw ExampleError.updateFunctionConfigurationError
            }

            guard let retVariables = response.variables else {
                throw ExampleError.environmentVariablesMissingError
            }

            for envVar in retVariables {
                if envVar.key == "LOG_LEVEL" && envVar.value != "DEBUG" {
                    print("*** Log level is not set to DEBUG!")
                    throw ExampleError.logLevelIncorrectError
                }
            }
        } catch {
            throw ExampleError.updateFunctionConfigurationError
        }
    }
```

- Pour plus de détails sur l'API, reportez-vous UpdateFunctionConfigurationà la section AWS SDK pour la référence de l'API Swift.

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Scénarios d'utilisation de Lambda AWS SDKs

Les exemples de code suivants vous montrent comment implémenter des scénarios courants dans Lambda avec. AWS SDKs Ces scénarios vous montrent comment accomplir des tâches spécifiques en appelant plusieurs fonctions au sein de Lambda ou combinées avec d'autres Services AWS. Chaque exemple inclut un lien vers le code source complet, où vous trouverez des instructions sur la configuration et l'exécution du code.

Les scénarios ciblent un niveau d'expérience intermédiaire pour vous aider à comprendre les actions de service dans leur contexte.

Exemples

- Confirmez automatiquement les utilisateurs Amazon Cognito connus à l'aide d'une fonction Lambda à l'aide d'un SDK AWS
- Migrez automatiquement les utilisateurs connus d'Amazon Cognito à l'aide d'une fonction Lambda à l'aide d'un SDK AWS
- Créer une API REST API Gateway pour suivre les données de la COVID-19
- Créer une API REST de bibliothèque de prêt
- Créer une application de messagerie avec Step Functions
- Création d'une application de gestion des ressources photographiques permettant aux utilisateurs de gérer les photos à l'aide d'étiquettes
- Créer une application de chat Websocket avec API Gateway
- Créez une application qui analyse les commentaires des clients et synthétise le son
- Invoquer une fonction Lambda à partir d'un navigateur
- Transformation des données pour votre application avec S3 Object Lambda
- Utiliser API Gateway pour invoquer une fonction Lambda
- Utiliser les fonctions Step Functions pour invoquer des fonctions Lambda
- Utilisent des événements planifiés pour invoquer une fonction Lambda
- Utiliser l'API Amazon Neptune pour développer une fonction Lambda qui interroge les données d'un graphe

- Rédigez des données d'activité personnalisées à l'aide d'une fonction Lambda après l'authentification de l'utilisateur Amazon Cognito à l'aide d'un SDK AWS

Confirmez automatiquement les utilisateurs Amazon Cognito connus à l'aide d'une fonction Lambda à l'aide d'un SDK AWS

Les exemples de code suivants illustrent comment confirmer automatiquement les utilisateurs Amazon Cognito connus avec une fonction Lambda.

- Configurez un groupe d'utilisateurs pour appeler une fonction Lambda pour le déclencheur `PreSignUp`.
- Inscription d'un utilisateur avec Amazon Cognito.
- La fonction Lambda analyse une table DynamoDB et confirme automatiquement les utilisateurs connus.
- Connectez-vous en tant que nouvel utilisateur, puis nettoyez les ressources.

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

Exécutez un scénario interactif à une invite de commande.

```
import (
 "context"
 "errors"
 "log"
 "strings"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
```

```
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// AutoConfirm separates the steps of this scenario into individual functions so
 that
// they are simpler to read and understand.
type AutoConfirm struct {
 helper       IScenarioHelper
 questioner   demotools.IQuestioner
 resources    Resources
 cognitoActor *actions.CognitoActions
}

// NewAutoConfirm constructs a new auto confirm runner.
func NewAutoConfirm(sdkConfig aws.Config, questioner demotools.IQuestioner,
 helper IScenarioHelper) AutoConfirm {
 scenario := AutoConfirm{
  helper:       helper,
  questioner:   questioner,
  resources:    Resources{},
  cognitoActor: &actions.CognitoActions{CognitoClient:
 cognitoidentityprovider.NewFromConfig(sdkConfig)},
 }
 scenario.resources.init(scenario.cognitoActor, questioner)
 return scenario
}

// AddPreSignUpTrigger adds a Lambda handler as an invocation target for the
 PreSignUp trigger.
func (runner *AutoConfirm) AddPreSignUpTrigger(ctx context.Context, userPoolId
 string, functionArn string) {
 log.Printf("Let's add a Lambda function to handle the PreSignUp trigger from
 Cognito.\n" +
  "This trigger happens when a user signs up, and lets your function take action
 before the main Cognito\n" +
  "sign up processing occurs.\n")
 err := runner.cognitoActor.UpdateTriggers(
  ctx, userPoolId,
  actions.TriggerInfo{Trigger: actions.PreSignUp, HandlerArn:
 aws.String(functionArn)})
 if err != nil {
  panic(err)
 }
```

```
 log.Printf("Lambda function %v added to user pool %v to handle the PreSignUp
 trigger.\n",
  functionArn, userPoolId)
}

// SignUpUser signs up a user from the known user table with a password you
 specify.
func (runner *AutoConfirm) SignUpUser(ctx context.Context, clientId string,
 usersTable string) (string, string) {
 log.Println("Let's sign up a user to your Cognito user pool. When the user's
 email matches an email in the\n" +
  "DynamoDB known users table, it is automatically verified and the user is
 confirmed.")

 knownUsers, err := runner.helper.GetKnownUsers(ctx, usersTable)
 if err != nil {
  panic(err)
 }
 userChoice := runner.questioner.AskChoice("Which user do you want to use?\n",
 knownUsers.UserNameList())
 user := knownUsers.Users[userChoice]

 var signedUp bool
 var userConfirmed bool
 password := runner.questioner.AskPassword("Enter a password that has at least
 eight characters, uppercase, lowercase, numbers and symbols.\n"+
  "(the password will not display as you type):", 8)
 for !signedUp {
  log.Printf("Signing up user '%v' with email '%v' to Cognito.\n", user.UserName,
 user.UserEmail)
  userConfirmed, err = runner.cognitoActor.SignUp(ctx, clientId, user.UserName,
 password, user.UserEmail)
  if err != nil {
   var invalidPassword *types.InvalidPasswordException
   if errors.As(err, &invalidPassword) {
    password = runner.questioner.AskPassword("Enter another password:", 8)
   } else {
    panic(err)
   }
  } else {
   signedUp = true
  }
 }
 log.Printf("User %v signed up, confirmed = %v.\n", user.UserName, userConfirmed)
```

```
 log.Println(strings.Repeat("-", 88))

 return user.UserName, password
}

// SignInUser signs in a user.
func (runner *AutoConfirm) SignInUser(ctx context.Context, clientId string,
 userName string, password string) string {
 runner.questioner.Ask("Press Enter when you're ready to continue.")
 log.Printf("Let's sign in as %v...\n", userName)
 authResult, err := runner.cognitoActor.SignIn(ctx, clientId, userName, password)
 if err != nil {
  panic(err)
 }
 log.Printf("Successfully signed in. Your access token starts with: %v...\n",
 (*authResult.AccessToken)[:10])
 log.Println(strings.Repeat("-", 88))
 return *authResult.AccessToken
}

// Run runs the scenario.
func (runner *AutoConfirm) Run(ctx context.Context, stackName string) {
 defer func() {
  if r := recover(); r != nil {
   log.Println("Something went wrong with the demo.")
   runner.resources.Cleanup(ctx)
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Printf("Welcome\n")

 log.Println(strings.Repeat("-", 88))

 stackOutputs, err := runner.helper.GetStackOutputs(ctx, stackName)
 if err != nil {
  panic(err)
 }
 runner.resources.userPoolId = stackOutputs["UserPoolId"]
 runner.helper.PopulateUserTable(ctx, stackOutputs["TableName"])

 runner.AddPreSignUpTrigger(ctx, stackOutputs["UserPoolId"],
 stackOutputs["AutoConfirmFunctionArn"])
```

```
 runner.resources.triggers = append(runner.resources.triggers, actions.PreSignUp)
 userName, password := runner.SignUpUser(ctx, stackOutputs["UserPoolClientId"],
 stackOutputs["TableName"])
 runner.helper.ListRecentLogEvents(ctx, stackOutputs["AutoConfirmFunction"])
 runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
  runner.SignInUser(ctx, stackOutputs["UserPoolClientId"], userName, password))

 runner.resources.Cleanup(ctx)

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

Gérez le déclencheur `PreSignUp` avec une fonction Lambda.

```
import (
 "context"
 "log"
 "os"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 dynamodbtypes "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

const TABLE_NAME = "TABLE_NAME"

// UserInfo defines structured user data that can be marshalled to a DynamoDB
 format.
type UserInfo struct {
 UserName  string `dynamodbav:"UserName"`
 UserEmail string `dynamodbav:"UserEmail"`
}

// GetKey marshals the user email value to a DynamoDB key format.
```

```
func (user UserInfo) GetKey() map[string]dynamodbtypes.AttributeValue {
 userEmail, err := attributevalue.Marshal(user.UserEmail)
 if err != nil {
  panic(err)
 }
 return map[string]dynamodbtypes.AttributeValue{"UserEmail": userEmail}
}

type handler struct {
 dynamoClient *dynamodb.Client
}

// HandleRequest handles the PreSignUp event by looking up a user in an Amazon
 DynamoDB table and
// specifying whether they should be confirmed and verified.
func (h *handler) HandleRequest(ctx context.Context, event
 events.CognitoEventUserPoolsPreSignup) (events.CognitoEventUserPoolsPreSignup,
 error) {
 log.Printf("Received presignup from %v for user '%v'", event.TriggerSource,
 event.UserName)
 if event.TriggerSource != "PreSignUp_SignUp" {
  // Other trigger sources, such as PreSignUp_AdminInitiateAuth, ignore the
 response from this handler.
  return event, nil
 }
 tableName := os.Getenv(TABLE_NAME)
 user := UserInfo{
  UserEmail: event.Request.UserAttributes["email"],
 }
 log.Printf("Looking up email %v in table %v.\n", user.UserEmail, tableName)
 output, err := h.dynamoClient.GetItem(ctx, &dynamodb.GetItemInput{
  Key:       user.GetKey(),
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Error looking up email %v.\n", user.UserEmail)
  return event, err
 }
 if output.Item == nil {
  log.Printf("Email %v not found. Email verification is required.\n",
 user.UserEmail)
  return event, err
 }
```

```
 err = attributevalue.UnmarshalMap(output.Item, &user)
 if err != nil {
  log.Printf("Couldn't unmarshal DynamoDB item. Here's why: %v\n", err)
  return event, err
 }

 if user.UserName != event.UserName {
  log.Printf("UserEmail %v found, but stored UserName '%v' does not match
 supplied UserName '%v'. Verification is required.\n",
   user.UserEmail, user.UserName, event.UserName)
 } else {
  log.Printf("UserEmail %v found with matching UserName %v. User is confirmed.
\n", user.UserEmail, user.UserName)
  event.Response.AutoConfirmUser = true
  event.Response.AutoVerifyEmail = true
 }

 return event, err
}

func main() {
 ctx := context.Background()
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Panicln(err)
 }
 h := handler{
  dynamoClient: dynamodb.NewFromConfig(sdkConfig),
 }
 lambda.Start(h.HandleRequest)
}
```

Créez une structure qui exécute les tâches courantes.

```
import (
 "context"
 "log"
 "strings"
 "time"
 "user_pools_and_lambda_triggers/actions"
```

```
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudformation"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// IScenarioHelper defines common functions used by the workflows in this
 example.
type IScenarioHelper interface {
 Pause(secs int)
 GetStackOutputs(ctx context.Context, stackName string) (actions.StackOutputs,
 error)
 PopulateUserTable(ctx context.Context, tableName string)
 GetKnownUsers(ctx context.Context, tableName string) (actions.UserList, error)
 AddKnownUser(ctx context.Context, tableName string, user actions.User)
 ListRecentLogEvents(ctx context.Context, functionName string)
}

// ScenarioHelper contains AWS wrapper structs used by the workflows in this
 example.
type ScenarioHelper struct {
 questioner  demotools.IQuestioner
 dynamoActor *actions.DynamoActions
 cfnActor    *actions.CloudFormationActions
 cwlActor    *actions.CloudWatchLogsActions
 isTestRun   bool
}

// NewScenarioHelper constructs a new scenario helper.
func NewScenarioHelper(sdkConfig aws.Config, questioner demotools.IQuestioner)
 ScenarioHelper {
 scenario := ScenarioHelper{
  questioner:  questioner,
  dynamoActor: &actions.DynamoActions{DynamoClient:
 dynamodb.NewFromConfig(sdkConfig)},
  cfnActor:    &actions.CloudFormationActions{CfnClient:
 cloudformation.NewFromConfig(sdkConfig)},
  cwlActor:    &actions.CloudWatchLogsActions{CwlClient:
 cloudwatchlogs.NewFromConfig(sdkConfig)},
 }
 return scenario
}
```

```
// Pause waits for the specified number of seconds.
func (helper ScenarioHelper) Pause(secs int) {
 if !helper.isTestRun {
  time.Sleep(time.Duration(secs) * time.Second)
 }
}

// GetStackOutputs gets the outputs from the specified CloudFormation stack in a
 structured format.
func (helper ScenarioHelper) GetStackOutputs(ctx context.Context, stackName
 string) (actions.StackOutputs, error) {
 return helper.cfnActor.GetOutputs(ctx, stackName), nil
}

// PopulateUserTable fills the known user table with example data.
func (helper ScenarioHelper) PopulateUserTable(ctx context.Context, tableName
 string) {
 log.Printf("First, let's add some users to the DynamoDB %v table we'll use for
 this example.\n", tableName)
 err := helper.dynamoActor.PopulateTable(ctx, tableName)
 if err != nil {
  panic(err)
 }
}

// GetKnownUsers gets the users from the known users table in a structured
 format.
func (helper ScenarioHelper) GetKnownUsers(ctx context.Context, tableName string)
 (actions.UserList, error) {
 knownUsers, err := helper.dynamoActor.Scan(ctx, tableName)
 if err != nil {
  log.Printf("Couldn't get known users from table %v. Here's why: %v\n",
 tableName, err)
 }
 return knownUsers, err
}

// AddKnownUser adds a user to the known users table.
func (helper ScenarioHelper) AddKnownUser(ctx context.Context, tableName string,
 user actions.User) {
 log.Printf("Adding user '%v' with email '%v' to the DynamoDB known users
 table...\n",
  user.UserName, user.UserEmail)
```

```
 err := helper.dynamoActor.AddUser(ctx, tableName, user)
 if err != nil {
  panic(err)
 }
}

// ListRecentLogEvents gets the most recent log stream and events for the
 specified Lambda function and displays them.
func (helper ScenarioHelper) ListRecentLogEvents(ctx context.Context,
 functionName string) {
 log.Println("Waiting a few seconds to let Lambda write to CloudWatch Logs...")
 helper.Pause(10)
 log.Println("Okay, let's check the logs to find what's happened recently with
 your Lambda function.")
 logStream, err := helper.cwlActor.GetLatestLogStream(ctx, functionName)
 if err != nil {
  panic(err)
 }
 log.Printf("Getting some recent events from log stream %v\n",
 *logStream.LogStreamName)
 events, err := helper.cwlActor.GetLogEvents(ctx, functionName,
 *logStream.LogStreamName, 10)
 if err != nil {
  panic(err)
 }
 for _, event := range events {
  log.Printf("\t%v", *event.Message)
 }
 log.Println(strings.Repeat("-", 88))
}
```

Créez une structure qui encapsule les actions Amazon Cognito.

```
import (
 "context"
 "errors"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
```

```
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
)

type CognitoActions struct {
 CognitoClient *cognitoidentityprovider.Client
}



// Trigger and TriggerInfo define typed data for updating an Amazon Cognito
 trigger.
type Trigger int

const (
 PreSignUp Trigger = iota
 UserMigration
 PostAuthentication
)

type TriggerInfo struct {
 Trigger    Trigger
 HandlerArn *string
}

// UpdateTriggers adds or removes Lambda triggers for a user pool. When a trigger
 is specified with a `nil` value,
// it is removed from the user pool.
func (actor CognitoActions) UpdateTriggers(ctx context.Context, userPoolId
 string, triggers ...TriggerInfo) error {
 output, err := actor.CognitoClient.DescribeUserPool(ctx,
 &cognitoidentityprovider.DescribeUserPoolInput{
  UserPoolId: aws.String(userPoolId),
 })
 if err != nil {
  log.Printf("Couldn't get info about user pool %v. Here's why: %v\n",
 userPoolId, err)
  return err
 }
 lambdaConfig := output.UserPool.LambdaConfig
 for _, trigger := range triggers {
  switch trigger.Trigger {
  case PreSignUp:
   lambdaConfig.PreSignUp = trigger.HandlerArn
  case UserMigration:
```

```
   lambdaConfig.UserMigration = trigger.HandlerArn
  case PostAuthentication:
   lambdaConfig.PostAuthentication = trigger.HandlerArn
  }
 }
 _, err = actor.CognitoClient.UpdateUserPool(ctx,
 &cognitoidentityprovider.UpdateUserPoolInput{
  UserPoolId:   aws.String(userPoolId),
  LambdaConfig: lambdaConfig,
 })
 if err != nil {
  log.Printf("Couldn't update user pool %v. Here's why: %v\n", userPoolId, err)
 }
 return err
}



// SignUp signs up a user with Amazon Cognito.
func (actor CognitoActions) SignUp(ctx context.Context, clientId string, userName
 string, password string, userEmail string) (bool, error) {
 confirmed := false
 output, err := actor.CognitoClient.SignUp(ctx,
 &cognitoidentityprovider.SignUpInput{
  ClientId: aws.String(clientId),
  Password: aws.String(password),
  Username: aws.String(userName),
  UserAttributes: []types.AttributeType{
   {Name: aws.String("email"), Value: aws.String(userEmail)},
  },
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't sign up user %v. Here's why: %v\n", userName, err)
  }
 } else {
  confirmed = output.UserConfirmed
 }
 return confirmed, err
}
```

```
// SignIn signs in a user to Amazon Cognito using a username and password
 authentication flow.
func (actor CognitoActions) SignIn(ctx context.Context, clientId string, userName
 string, password string) (*types.AuthenticationResultType, error) {
 var authResult *types.AuthenticationResultType
 output, err := actor.CognitoClient.InitiateAuth(ctx,
 &cognitoidentityprovider.InitiateAuthInput{
  AuthFlow:       "USER_PASSWORD_AUTH",
  ClientId:       aws.String(clientId),
  AuthParameters: map[string]string{"USERNAME": userName, "PASSWORD": password},
 })
 if err != nil {
  var resetRequired *types.PasswordResetRequiredException
  if errors.As(err, &resetRequired) {
   log.Println(*resetRequired.Message)
  } else {
   log.Printf("Couldn't sign in user %v. Here's why: %v\n", userName, err)
  }
 } else {
  authResult = output.AuthenticationResult
 }
 return authResult, err
}



// ForgotPassword starts a password recovery flow for a user. This flow typically
 sends a confirmation code
// to the user's configured notification destination, such as email.
func (actor CognitoActions) ForgotPassword(ctx context.Context, clientId string,
 userName string) (*types.CodeDeliveryDetailsType, error) {
 output, err := actor.CognitoClient.ForgotPassword(ctx,
 &cognitoidentityprovider.ForgotPasswordInput{
  ClientId: aws.String(clientId),
  Username: aws.String(userName),
 })
 if err != nil {
  log.Printf("Couldn't start password reset for user '%v'. Here;s why: %v\n",
 userName, err)
 }
 return output.CodeDeliveryDetails, err
}
```

```
// ConfirmForgotPassword confirms a user with a confirmation code and a new
 password.
func (actor CognitoActions) ConfirmForgotPassword(ctx context.Context, clientId
 string, code string, userName string, password string) error {
 _, err := actor.CognitoClient.ConfirmForgotPassword(ctx,
 &cognitoidentityprovider.ConfirmForgotPasswordInput{
  ClientId:         aws.String(clientId),
  ConfirmationCode: aws.String(code),
  Password:         aws.String(password),
  Username:         aws.String(userName),
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't confirm user %v. Here's why: %v", userName, err)
  }
 }
 return err
}



// DeleteUser removes a user from the user pool.
func (actor CognitoActions) DeleteUser(ctx context.Context, userAccessToken
 string) error {
 _, err := actor.CognitoClient.DeleteUser(ctx,
 &cognitoidentityprovider.DeleteUserInput{
  AccessToken: aws.String(userAccessToken),
 })
 if err != nil {
  log.Printf("Couldn't delete user. Here's why: %v\n", err)
 }
 return err
}



// AdminCreateUser uses administrator credentials to add a user to a user pool.
 This method leaves the user
```

```
// in a state that requires they enter a new password next time they sign in.
func (actor CognitoActions) AdminCreateUser(ctx context.Context, userPoolId
 string, userName string, userEmail string) error {
 _, err := actor.CognitoClient.AdminCreateUser(ctx,
 &cognitoidentityprovider.AdminCreateUserInput{
  UserPoolId:     aws.String(userPoolId),
  Username:       aws.String(userName),
  MessageAction:  types.MessageActionTypeSuppress,
  UserAttributes: []types.AttributeType{{Name: aws.String("email"), Value:
 aws.String(userEmail)}},
 })
 if err != nil {
  var userExists *types.UsernameExistsException
  if errors.As(err, &userExists) {
   log.Printf("User %v already exists in the user pool.", userName)
   err = nil
  } else {
   log.Printf("Couldn't create user %v. Here's why: %v\n", userName, err)
  }
 }
 return err
}



// AdminSetUserPassword uses administrator credentials to set a password for a
 user without requiring a
// temporary password.
func (actor CognitoActions) AdminSetUserPassword(ctx context.Context, userPoolId
 string, userName string, password string) error {
 _, err := actor.CognitoClient.AdminSetUserPassword(ctx,
 &cognitoidentityprovider.AdminSetUserPasswordInput{
  Password:   aws.String(password),
  UserPoolId: aws.String(userPoolId),
  Username:   aws.String(userName),
  Permanent:  true,
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't set password for user %v. Here's why: %v\n", userName,
 err)
```

```
  }
 }
 return err
}
```

Créez une structure qui encapsule les actions DynamoDB.

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// DynamoActions encapsulates the Amazon Simple Notification Service (Amazon SNS)
 actions
// used in the examples.
type DynamoActions struct {
 DynamoClient *dynamodb.Client
}

// User defines structured user data.
type User struct {
 UserName  string
 UserEmail string
 LastLogin *LoginInfo `dynamodbav:",omitempty"`
}

// LoginInfo defines structured custom login data.
type LoginInfo struct {
 UserPoolId string
 ClientId   string
 Time       string
}

// UserList defines a list of users.
```

```
type UserList struct {
 Users []User
}

// UserNameList returns the usernames contained in a UserList as a list of
 strings.
func (users *UserList) UserNameList() []string {
 names := make([]string, len(users.Users))
 for i := 0; i < len(users.Users); i++ {
  names[i] = users.Users[i].UserName
 }
 return names
}

// PopulateTable adds a set of test users to the table.
func (actor DynamoActions) PopulateTable(ctx context.Context, tableName string)
 error {
 var err error
 var item map[string]types.AttributeValue
 var writeReqs []types.WriteRequest
 for i := 1; i < 4; i++ {
  item, err = attributevalue.MarshalMap(User{UserName: fmt.Sprintf("test_user_
%v", i), UserEmail: fmt.Sprintf("test_email_%v@example.com", i)})
  if err != nil {
   log.Printf("Couldn't marshall user into DynamoDB format. Here's why: %v\n",
 err)
   return err
  }
  writeReqs = append(writeReqs, types.WriteRequest{PutRequest:
 &types.PutRequest{Item: item}})
 }
 _, err = actor.DynamoClient.BatchWriteItem(ctx, &dynamodb.BatchWriteItemInput{
  RequestItems: map[string][]types.WriteRequest{tableName: writeReqs},
 })
 if err != nil {
  log.Printf("Couldn't populate table %v with users. Here's why: %v\n",
 tableName, err)
 }
 return err
}

// Scan scans the table for all items.
func (actor DynamoActions) Scan(ctx context.Context, tableName string) (UserList,
 error) {
```

```
 var userList UserList
 output, err := actor.DynamoClient.Scan(ctx, &dynamodb.ScanInput{
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't scan table %v for items. Here's why: %v\n", tableName,
 err)
 } else {
  err = attributevalue.UnmarshalListOfMaps(output.Items, &userList.Users)
  if err != nil {
   log.Printf("Couldn't unmarshal items into users. Here's why: %v\n", err)
  }
 }
 return userList, err
}

// AddUser adds a user item to a table.
func (actor DynamoActions) AddUser(ctx context.Context, tableName string, user
 User) error {
 userItem, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshall user to item. Here's why: %v\n", err)
 }
 _, err = actor.DynamoClient.PutItem(ctx, &dynamodb.PutItemInput{
  Item:      userItem,
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't put item in table %v. Here's why: %v", tableName, err)
 }
 return err
}
```

Créez une structure qui englobe les actions CloudWatch Logs.

```
import (
 "context"
 "fmt"
 "log"
```

```
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs/types"
)

type CloudWatchLogsActions struct {
 CwlClient *cloudwatchlogs.Client
}

// GetLatestLogStream gets the most recent log stream for a Lambda function.
func (actor CloudWatchLogsActions) GetLatestLogStream(ctx context.Context,
 functionName string) (types.LogStream, error) {
 var logStream types.LogStream
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.DescribeLogStreams(ctx,
 &cloudwatchlogs.DescribeLogStreamsInput{
  Descending:   aws.Bool(true),
  Limit:        aws.Int32(1),
  LogGroupName: aws.String(logGroupName),
  OrderBy:      types.OrderByLastEventTime,
 })
 if err != nil {
  log.Printf("Couldn't get log streams for log group %v. Here's why: %v\n",
 logGroupName, err)
 } else {
  logStream = output.LogStreams[0]
 }
 return logStream, err
}

// GetLogEvents gets the most recent eventCount events from the specified log
 stream.
func (actor CloudWatchLogsActions) GetLogEvents(ctx context.Context, functionName
 string, logStreamName string, eventCount int32) (
 []types.OutputLogEvent, error) {
 var events []types.OutputLogEvent
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.GetLogEvents(ctx,
 &cloudwatchlogs.GetLogEventsInput{
  LogStreamName: aws.String(logStreamName),
  Limit:         aws.Int32(eventCount),
  LogGroupName:  aws.String(logGroupName),
 })
 if err != nil {
```

```
  log.Printf("Couldn't get log event for log stream %v. Here's why: %v\n",
 logStreamName, err)
 } else {
  events = output.Events
 }
 return events, err
}
```

Créez une structure qui englobe les actions. CloudFormation

```
import (
 "context"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudformation"
)

// StackOutputs defines a map of outputs from a specific stack.
type StackOutputs map[string]string

type CloudFormationActions struct {
 CfnClient *cloudformation.Client
}

// GetOutputs gets the outputs from a CloudFormation stack and puts them into a
 structured format.
func (actor CloudFormationActions) GetOutputs(ctx context.Context, stackName
 string) StackOutputs {
 output, err := actor.CfnClient.DescribeStacks(ctx,
 &cloudformation.DescribeStacksInput{
  StackName: aws.String(stackName),
 })
 if err != nil || len(output.Stacks) == 0 {
  log.Panicf("Couldn't find a CloudFormation stack named %v. Here's why: %v\n",
 stackName, err)
 }
 stackOutputs := StackOutputs{}
 for _, out := range output.Stacks[0].Outputs {
  stackOutputs[*out.OutputKey] = *out.OutputValue
```

```
 }
 return stackOutputs
}
```

Nettoyez les ressources.

```
import (
 "context"
 "log"
 "user_pools_and_lambda_triggers/actions"

 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// Resources keeps track of AWS resources created during an example and handles
// cleanup when the example finishes.
type Resources struct {
 userPoolId       string
 userAccessTokens []string
 triggers         []actions.Trigger

 cognitoActor *actions.CognitoActions
 questioner   demotools.IQuestioner
}

func (resources *Resources) init(cognitoActor *actions.CognitoActions, questioner
 demotools.IQuestioner) {
 resources.userAccessTokens = []string{}
 resources.triggers = []actions.Trigger{}
 resources.cognitoActor = cognitoActor
 resources.questioner = questioner
}

// Cleanup deletes all AWS resources created during an example.
func (resources *Resources) Cleanup(ctx context.Context) {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong during cleanup.\n%v\n", r)
   log.Println("Use the AWS Management Console to remove any remaining resources
 \n" +
```

```
    "that were created for this scenario.")
  }
 }()

 wantDelete := resources.questioner.AskBool("Do you want to remove all of the AWS
 resources that were created "+
  "during this demo (y/n)?", "y")
 if wantDelete {
  for _, accessToken := range resources.userAccessTokens {
   err := resources.cognitoActor.DeleteUser(ctx, accessToken)
   if err != nil {
    log.Println("Couldn't delete user during cleanup.")
    panic(err)
   }
   log.Println("Deleted user.")
  }
  triggerList := make([]actions.TriggerInfo, len(resources.triggers))
  for i := 0; i < len(resources.triggers); i++ {
   triggerList[i] = actions.TriggerInfo{Trigger: resources.triggers[i],
 HandlerArn: nil}
  }
  err := resources.cognitoActor.UpdateTriggers(ctx, resources.userPoolId,
 triggerList...)
  if err != nil {
   log.Println("Couldn't update Cognito triggers during cleanup.")
   panic(err)
  }
  log.Println("Removed Cognito triggers from user pool.")
 } else {
  log.Println("Be sure to remove resources when you're done with them to avoid
 unexpected charges!")
 }
}
```

- Pour plus de détails sur l'API, consultez les rubriques suivantes dans la Référence des API du kit AWS SDK pour Go .
  - DeleteUser
  - InitiateAuth
  - SignUp
  - UpdateUserPool

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

Configurez une exécution interactive de type « Scénario ». Les exemples JavaScript (v3) partagent un générateur de scénarios pour rationaliser les exemples complexes. Le code source complet est activé GitHub.

```
import { AutoConfirm } from "./scenario-auto-confirm.js";

/**
 * The context is passed to every scenario. Scenario steps
 * will modify the context.
 */
const context = {
  errors: [],
  users: [
    {
      UserName: "test_user_1",
      UserEmail: "test_email_1@example.com",
    },
    {
      UserName: "test_user_2",
      UserEmail: "test_email_2@example.com",
    },
    {
      UserName: "test_user_3",
      UserEmail: "test_email_3@example.com",
    },
  ],
};

/**
 * Three Scenarios are created for the workflow. A Scenario is an orchestration
 class
 * that simplifies running a series of steps.
```

```
 */
export const scenarios = {
  // Demonstrate automatically confirming known users in a database.
  "auto-confirm": AutoConfirm(context),
};

// Call function if run directly
import { fileURLToPath } from "node:url";
import { parseScenarioArgs } from "@aws-doc-sdk-examples/lib/scenario/index.js";

if (process.argv[1] === fileURLToPath(import.meta.url)) {
  parseScenarioArgs(scenarios, {
    name: "Cognito user pools and triggers",
    description:
      "Demonstrate how to use the AWS SDKs to customize Amazon Cognito
 authentication behavior.",
  });
}
```

Ce scénario illustre la confirmation automatique d'un utilisateur connu. Il orchestre les étapes de l'exemple.

```
import { wait } from "@aws-doc-sdk-examples/lib/utils/util-timers.js";
import {
  Scenario,
  ScenarioAction,
  ScenarioInput,
  ScenarioOutput,
} from "@aws-doc-sdk-examples/lib/scenario/scenario.js";

import {
  getStackOutputs,
  logCleanUpReminder,
  promptForStackName,
  promptForStackRegion,
  skipWhenErrors,
} from "./steps-common.js";
import { populateTable } from "./actions/dynamodb-actions.js";
import {
  addPreSignUpHandler,
  deleteUser,
  getUser,
```

```
  signIn,
  signUpUser,
} from "./actions/cognito-actions.js";
import {
  getLatestLogStreamForLambda,
  getLogEvents,
} from "./actions/cloudwatch-logs-actions.js";

/**
 * @typedef {{
 *   errors: Error[],
 *   password: string,
 *   users: { UserName: string, UserEmail: string }[],
 *   selectedUser?: string,
 *   stackName?: string,
 *   stackRegion?: string,
 *   token?: string,
 *   confirmDeleteSignedInUser?: boolean,
 *   TableName?: string,
 *   UserPoolClientId?: string,
 *   UserPoolId?: string,
 *   UserPoolArn?: string,
 *   AutoConfirmHandlerArn?: string,
 *   AutoConfirmHandlerName?: string
 * }} State
 */

const greeting = new ScenarioOutput(
  "greeting",
  (/** @type {State} */ state) => `This demo will populate some users into the \
database created as part of the "${state.stackName}" stack. \
Then the AutoConfirmHandler will be linked to the PreSignUp \
trigger from Cognito. Finally, you will choose a user to sign up.`,
  { skipWhen: skipWhenErrors },
);

const logPopulatingUsers = new ScenarioOutput(
  "logPopulatingUsers",
  "Populating the DynamoDB table with some users.",
  { skipWhenErrors: skipWhenErrors },
);

const logPopulatingUsersComplete = new ScenarioOutput(
  "logPopulatingUsersComplete",
```

```
  "Done populating users.",
  { skipWhen: skipWhenErrors },
);

const populateUsers = new ScenarioAction(
  "populateUsers",
  async (/** @type {State} */ state) => {
    const [_, err] = await populateTable({
      region: state.stackRegion,
      tableName: state.TableName,
      items: state.users,
    });
    if (err) {
      state.errors.push(err);
    }
  },
  {
    skipWhen: skipWhenErrors,
  },
);

const logSetupSignUpTrigger = new ScenarioOutput(
  "logSetupSignUpTrigger",
  "Setting up the PreSignUp trigger for the Cognito User Pool.",
  { skipWhen: skipWhenErrors },
);

const setupSignUpTrigger = new ScenarioAction(
  "setupSignUpTrigger",
  async (/** @type {State} */ state) => {
    const [_, err] = await addPreSignUpHandler({
      region: state.stackRegion,
      userPoolId: state.UserPoolId,
      handlerArn: state.AutoConfirmHandlerArn,
    });
    if (err) {
      state.errors.push(err);
    }
  },
  {
    skipWhen: skipWhenErrors,
  },
);
```

```
const logSetupSignUpTriggerComplete = new ScenarioOutput(
  "logSetupSignUpTriggerComplete",
  (
    /** @type {State} */ state,
  ) => `The lambda function "${state.AutoConfirmHandlerName}" \
has been configured as the PreSignUp trigger handler for the user pool
 "${state.UserPoolId}".`,
  { skipWhen: skipWhenErrors },
);

const selectUser = new ScenarioInput(
  "selectedUser",
  "Select a user to sign up.",
  {
    type: "select",
    choices: (/** @type {State} */ state) => state.users.map((u) => u.UserName),
    skipWhen: skipWhenErrors,
    default: (/** @type {State} */ state) => state.users[0].UserName,
  },
);

const checkIfUserAlreadyExists = new ScenarioAction(
  "checkIfUserAlreadyExists",
  async (/** @type {State} */ state) => {
    const [user, err] = await getUser({
      region: state.stackRegion,
      userPoolId: state.UserPoolId,
      username: state.selectedUser,
    });

    if (err?.name === "UserNotFoundException") {
      // Do nothing. We're not expecting the user to exist before
      // sign up is complete.
      return;
    }

    if (err) {
      state.errors.push(err);
      return;
    }

    if (user) {
      state.errors.push(
        new Error(
```

```
          `The user "${state.selectedUser}" already exists in the user pool
 "${state.UserPoolId}".`,
        ),
      );
    }
  },
  {
    skipWhen: skipWhenErrors,
  },
);

const createPassword = new ScenarioInput(
  "password",
  "Enter a password that has at least eight characters, uppercase, lowercase,
 numbers and symbols.",
  { type: "password", skipWhen: skipWhenErrors, default: "Abcd1234!" },
);

const logSignUpExistingUser = new ScenarioOutput(
  "logSignUpExistingUser",
  (/** @type {State} */ state) => `Signing up user "${state.selectedUser}".`,
  { skipWhen: skipWhenErrors },
);

const signUpExistingUser = new ScenarioAction(
  "signUpExistingUser",
  async (/** @type {State} */ state) => {
    const signUp = (password) =>
      signUpUser({
        region: state.stackRegion,
        userPoolClientId: state.UserPoolClientId,
        username: state.selectedUser,
        email: state.users.find((u) => u.UserName === state.selectedUser)
          .UserEmail,
        password,
      });

    let [_, err] = await signUp(state.password);

    while (err?.name === "InvalidPasswordException") {
      console.warn("The password you entered was invalid.");
      await createPassword.handle(state);
      [_, err] = await signUp(state.password);
    }
```

```
    if (err) {
      state.errors.push(err);
    }
  },
  { skipWhen: skipWhenErrors },
);

const logSignUpExistingUserComplete = new ScenarioOutput(
  "logSignUpExistingUserComplete",
  (/** @type {State} */ state) =>
    `"${state.selectedUser} was signed up successfully.`,
  { skipWhen: skipWhenErrors },
);

const logLambdaLogs = new ScenarioAction(
  "logLambdaLogs",
  async (/** @type {State} */ state) => {
    console.log(
      "Waiting a few seconds to let Lambda write to CloudWatch Logs...\n",
    );
    await wait(10);

    const [logStream, logStreamErr] = await getLatestLogStreamForLambda({
      functionName: state.AutoConfirmHandlerName,
      region: state.stackRegion,
    });
    if (logStreamErr) {
      state.errors.push(logStreamErr);
      return;
    }

    console.log(
      `Getting some recent events from log stream "${logStream.logStreamName}"`,
    );
    const [logEvents, logEventsErr] = await getLogEvents({
      functionName: state.AutoConfirmHandlerName,
      region: state.stackRegion,
      eventCount: 10,
      logStreamName: logStream.logStreamName,
    });
    if (logEventsErr) {
      state.errors.push(logEventsErr);
      return;
```

```
    }

    console.log(logEvents.map((ev) => `\t${ev.message}`).join(""));
  },
  { skipWhen: skipWhenErrors },
);

const logSignInUser = new ScenarioOutput(
  "logSignInUser",
  (/** @type {State} */ state) => `Let's sign in as ${state.selectedUser}`,
  { skipWhen: skipWhenErrors },
);

const signInUser = new ScenarioAction(
  "signInUser",
  async (/** @type {State} */ state) => {
    const [response, err] = await signIn({
      region: state.stackRegion,
      clientId: state.UserPoolClientId,
      username: state.selectedUser,
      password: state.password,
    });

    if (err?.name === "PasswordResetRequiredException") {
      state.errors.push(new Error("Please reset your password."));
      return;
    }

    if (err) {
      state.errors.push(err);
      return;
    }

    state.token = response?.AuthenticationResult?.AccessToken;
  },
  { skipWhen: skipWhenErrors },
);

const logSignInUserComplete = new ScenarioOutput(
  "logSignInUserComplete",
  (/** @type {State} */ state) =>
    `Successfully signed in. Your access token starts with:
 ${state.token.slice(0, 11)}`,
  { skipWhen: skipWhenErrors },
```

```
);

const confirmDeleteSignedInUser = new ScenarioInput(
  "confirmDeleteSignedInUser",
  "Do you want to delete the currently signed in user?",
  { type: "confirm", skipWhen: skipWhenErrors },
);

const deleteSignedInUser = new ScenarioAction(
  "deleteSignedInUser",
  async (/** @type {State} */ state) => {
    const [_, err] = await deleteUser({
      region: state.stackRegion,
      accessToken: state.token,
    });

    if (err) {
      state.errors.push(err);
    }
  },
  {
    skipWhen: (/** @type {State} */ state) =>
      skipWhenErrors(state) || !state.confirmDeleteSignedInUser,
  },
);

const logErrors = new ScenarioOutput(
  "logErrors",
  (/** @type {State}*/ state) => {
    const errorList = state.errors
      .map((err) => ` - ${err.name}: ${err.message}`)
      .join("\n");
    return `Scenario errors found:\n${errorList}`;
  },
  {
    // Don't log errors when there aren't any!
    skipWhen: (/** @type {State} */ state) => state.errors.length === 0,
  },
);

export const AutoConfirm = (context) =>
  new Scenario(
    "AutoConfirm",
    [
```

```
      promptForStackName,
      promptForStackRegion,
      getStackOutputs,
      greeting,
      logPopulatingUsers,
      populateUsers,
      logPopulatingUsersComplete,
      logSetupSignUpTrigger,
      setupSignUpTrigger,
      logSetupSignUpTriggerComplete,
      selectUser,
      checkIfUserAlreadyExists,
      createPassword,
      logSignUpExistingUser,
      signUpExistingUser,
      logSignUpExistingUserComplete,
      logLambdaLogs,
      logSignInUser,
      signInUser,
      logSignInUserComplete,
      confirmDeleteSignedInUser,
      deleteSignedInUser,
      logCleanUpReminder,
      logErrors,
    ],
    context,
  );
```

Ces étapes sont partagées avec d'autres scénarios.

```
import {
  ScenarioAction,
  ScenarioInput,
  ScenarioOutput,
} from "@aws-doc-sdk-examples/lib/scenario/scenario.js";
import { getCfnOutputs } from "@aws-doc-sdk-examples/lib/sdk/cfn-outputs.js";

export const skipWhenErrors = (state) => state.errors.length > 0;

export const getStackOutputs = new ScenarioAction(
  "getStackOutputs",
  async (state) => {
```

```
    if (!state.stackName || !state.stackRegion) {
      state.errors.push(
        new Error(
          "No stack name or region provided. The stack name and \
region are required to fetch CFN outputs relevant to this example.",
        ),
      );
      return;
    }

    const outputs = await getCfnOutputs(state.stackName, state.stackRegion);
    Object.assign(state, outputs);
  },
);

export const promptForStackName = new ScenarioInput(
  "stackName",
  "Enter the name of the stack you deployed earlier.",
  { type: "input", default: "PoolsAndTriggersStack" },
);

export const promptForStackRegion = new ScenarioInput(
  "stackRegion",
  "Enter the region of the stack you deployed earlier.",
  { type: "input", default: "us-east-1" },
);

export const logCleanUpReminder = new ScenarioOutput(
  "logCleanUpReminder",
  "All done. Remember to run 'cdk destroy' to teardown the stack.",
  { skipWhen: skipWhenErrors },
);
```

Un gestionnaire pour le déclencheur `PreSignUp` avec une fonction Lambda.

```
import type { PreSignUpTriggerEvent, Handler } from "aws-lambda";
import type { UserRepository } from "./user-repository";
import { DynamoDBUserRepository } from "./user-repository";

export class PreSignUpHandler {
  private userRepository: UserRepository;
```

```
  constructor(userRepository: UserRepository) {
    this.userRepository = userRepository;
  }

  private isPreSignUpTriggerSource(event: PreSignUpTriggerEvent): boolean {
    return event.triggerSource === "PreSignUp_SignUp";
  }

  private getEventUserEmail(event: PreSignUpTriggerEvent): string {
    return event.request.userAttributes.email;
  }

  async handlePreSignUpTriggerEvent(
    event: PreSignUpTriggerEvent,
  ): Promise<PreSignUpTriggerEvent> {
    console.log(
      `Received presignup from ${event.triggerSource} for user
'${event.userName}'`,
    );

    if (!this.isPreSignUpTriggerSource(event)) {
      return event;
    }

    const eventEmail = this.getEventUserEmail(event);
    console.log(`Looking up email ${eventEmail}.`);
    const storedUserInfo =
      await this.userRepository.getUserInfoByEmail(eventEmail);

    if (!storedUserInfo) {
      console.log(
        `Email ${eventEmail} not found. Email verification is required.`,
      );
      return event;
    }

    if (storedUserInfo.UserName !== event.userName) {
      console.log(
        `UserEmail ${eventEmail} found, but stored UserName
'${storedUserInfo.UserName}' does not match supplied UserName
'${event.userName}'. Verification is required.`,
      );
    } else {
      console.log(
```

```
        `UserEmail ${eventEmail} found with matching UserName
 ${storedUserInfo.UserName}. User is confirmed.`,
      );
      event.response.autoConfirmUser = true;
      event.response.autoVerifyEmail = true;
    }
    return event;
  }
}

const createPreSignUpHandler = (): PreSignUpHandler => {
  const tableName = process.env.TABLE_NAME;
  if (!tableName) {
    throw new Error("TABLE_NAME environment variable is not set");
  }

  const userRepository = new DynamoDBUserRepository(tableName);
  return new PreSignUpHandler(userRepository);
};

export const handler: Handler = async (event: PreSignUpTriggerEvent) => {
  const preSignUpHandler = createPreSignUpHandler();
  return preSignUpHandler.handlePreSignUpTriggerEvent(event);
};
```

Module d'actions de CloudWatch journalisation.

```
import {
  CloudWatchLogsClient,
  GetLogEventsCommand,
  OrderBy,
  paginateDescribeLogStreams,
} from "@aws-sdk/client-cloudwatch-logs";

/**
 * Get the latest log stream for a Lambda function.
 * @param {{ functionName: string, region: string }} config
 * @returns {Promise<[import("@aws-sdk/client-cloudwatch-logs").LogStream | null,
 unknown]>}
 */
export const getLatestLogStreamForLambda = async ({ functionName, region }) => {
```

```
  try {
    const logGroupName = `/aws/lambda/${functionName}`;
    const cwlClient = new CloudWatchLogsClient({ region });
    const paginator = paginateDescribeLogStreams(
      { client: cwlClient },
      {
        descending: true,
        limit: 1,
        orderBy: OrderBy.LastEventTime,
        logGroupName,
      },
    );

    for await (const page of paginator) {
      return [page.logStreams[0], null];
    }
  } catch (err) {
    return [null, err];
  }
};

/**
 * Get the log events for a Lambda function's log stream.
 * @param {{
 *   functionName: string,
 *   logStreamName: string,
 *   eventCount: number,
 *   region: string
 * }} config
 * @returns {Promise<[import("@aws-sdk/client-cloudwatch-logs").OutputLogEvent[]
 | null, unknown]>}
 */
export const getLogEvents = async ({
  functionName,
  logStreamName,
  eventCount,
  region,
}) => {
  try {
    const cwlClient = new CloudWatchLogsClient({ region });
    const logGroupName = `/aws/lambda/${functionName}`;
    const response = await cwlClient.send(
      new GetLogEventsCommand({
        logStreamName: logStreamName,
```

```
        limit: eventCount,
        logGroupName: logGroupName,
      }),
    );

    return [response.events, null];
  } catch (err) {
    return [null, err];
  }
};
```

Module d'actions Amazon Cognito.

```
import {
  AdminGetUserCommand,
  CognitoIdentityProviderClient,
  DeleteUserCommand,
  InitiateAuthCommand,
  SignUpCommand,
  UpdateUserPoolCommand,
} from "@aws-sdk/client-cognito-identity-provider";

/**
 * Connect a Lambda function to the PreSignUp trigger for a Cognito user pool
 * @param {{ region: string, userPoolId: string, handlerArn: string }} config
 * @returns {Promise<[import("@aws-sdk/client-cognito-identity-
provider").UpdateUserPoolCommandOutput | null, unknown]>}
 */
export const addPreSignUpHandler = async ({
  region,
  userPoolId,
  handlerArn,
}) => {
  try {
    const cognitoClient = new CognitoIdentityProviderClient({
      region,
    });

    const command = new UpdateUserPoolCommand({
      UserPoolId: userPoolId,
      LambdaConfig: {
```

```
        PreSignUp: handlerArn,
      },
    });

    const response = await cognitoClient.send(command);
    return [response, null];
  } catch (err) {
    return [null, err];
  }
};

/**
 * Attempt to register a user to a user pool with a given username and password.
 * @param {{
 *   region: string,
 *   userPoolClientId: string,
 *   username: string,
 *   email: string,
 *   password: string
 * }} config
 * @returns {Promise<[import("@aws-sdk/client-cognito-identity-
provider").SignUpCommandOutput | null, unknown]>}
 */
export const signUpUser = async ({
  region,
  userPoolClientId,
  username,
  email,
  password,
}) => {
  try {
    const cognitoClient = new CognitoIdentityProviderClient({
      region,
    });

    const response = await cognitoClient.send(
      new SignUpCommand({
        ClientId: userPoolClientId,
        Username: username,
        Password: password,
        UserAttributes: [{ Name: "email", Value: email }],
      }),
    );
    return [response, null];
```

```
  } catch (err) {
    return [null, err];
  }
};

/**
 * Sign in a user to Amazon Cognito using a username and password authentication
 flow.
 * @param {{ region: string, clientId: string, username: string, password:
 string }} config
 * @returns {Promise<[import("@aws-sdk/client-cognito-identity-
provider").InitiateAuthCommandOutput | null, unknown]>}
 */
export const signIn = async ({ region, clientId, username, password }) => {
  try {
    const cognitoClient = new CognitoIdentityProviderClient({ region });
    const response = await cognitoClient.send(
      new InitiateAuthCommand({
        AuthFlow: "USER_PASSWORD_AUTH",
        ClientId: clientId,
        AuthParameters: { USERNAME: username, PASSWORD: password },
      }),
    );
    return [response, null];
  } catch (err) {
    return [null, err];
  }
};

/**
 * Retrieve an existing user from a user pool.
 * @param {{ region: string, userPoolId: string, username: string }} config
 * @returns {Promise<[import("@aws-sdk/client-cognito-identity-
provider").AdminGetUserCommandOutput | null, unknown]>}
 */
export const getUser = async ({ region, userPoolId, username }) => {
  try {
    const cognitoClient = new CognitoIdentityProviderClient({ region });
    const response = await cognitoClient.send(
      new AdminGetUserCommand({
        UserPoolId: userPoolId,
        Username: username,
      }),
    );
```

```
    return [response, null];
  } catch (err) {
    return [null, err];
  }
};

/**
 * Delete the signed-in user. Useful for allowing a user to delete their
 * own profile.
 * @param {{ region: string, accessToken: string }} config
 * @returns {Promise<[import("@aws-sdk/client-cognito-identity-
provider").DeleteUserCommandOutput | null, unknown]>}
 */
export const deleteUser = async ({ region, accessToken }) => {
  try {
    const client = new CognitoIdentityProviderClient({ region });
    const response = await client.send(
      new DeleteUserCommand({ AccessToken: accessToken }),
    );
    return [response, null];
  } catch (err) {
    return [null, err];
  }
};
```

Module d’actions DynamoDB.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import {
  BatchWriteCommand,
  DynamoDBDocumentClient,
} from "@aws-sdk/lib-dynamodb";

/**
 * Populate a DynamoDB table with provide items.
 * @param {{ region: string, tableName: string, items: Record<string,
 unknown>[] }} config
 * @returns {Promise<[import("@aws-sdk/lib-dynamodb").BatchWriteCommandOutput |
 null, unknown]>}
 */
export const populateTable = async ({ region, tableName, items }) => {
```

```
  try {
    const ddbClient = new DynamoDBClient({ region });
    const docClient = DynamoDBDocumentClient.from(ddbClient);
    const response = await docClient.send(
      new BatchWriteCommand({
        RequestItems: {
          [tableName]: items.map((item) => ({
            PutRequest: {
              Item: item,
            },
          })),
        },
      }),
    );
    return [response, null];
  } catch (err) {
    return [null, err];
  }
};
```

- Pour plus de détails sur l'API, consultez les rubriques suivantes dans la Référence des API du kit AWS SDK pour JavaScript .
  - DeleteUser
  - InitiateAuth
  - SignUp
  - UpdateUserPool

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Migrez automatiquement les utilisateurs connus d'Amazon Cognito à l'aide d'une fonction Lambda à l'aide d'un SDK AWS

L'exemple de code suivant illustre comment effectuer automatiquement une migration des utilisateurs Amazon Cognito connus avec une fonction Lambda.

- Configurez un groupe d'utilisateurs pour appeler une fonction Lambda pour le déclencheur `MigrateUser`.
- Connectez-vous à Amazon Cognito avec un nom d'utilisateur et une adresse e-mail qui ne figurent pas dans le groupe d'utilisateurs.
- La fonction Lambda analyse une table DynamoDB et transfère automatiquement les utilisateurs connus vers le groupe d'utilisateurs.
- Exécutez le flux de mots de passe oubliés pour réinitialiser le mot de passe de l'utilisateur soumis à la migration.
- Connectez-vous en tant que nouvel utilisateur, puis nettoyez les ressources.

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

Exécutez un scénario interactif à une invite de commande.

```
import (
 "context"
 "errors"
 "fmt"
 "log"
 "strings"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// MigrateUser separates the steps of this scenario into individual functions so
 that
```

```
// they are simpler to read and understand.
type MigrateUser struct {
 helper       IScenarioHelper
 questioner   demotools.IQuestioner
 resources    Resources
 cognitoActor *actions.CognitoActions
}

// NewMigrateUser constructs a new migrate user runner.
func NewMigrateUser(sdkConfig aws.Config, questioner demotools.IQuestioner,
 helper IScenarioHelper) MigrateUser {
 scenario := MigrateUser{
  helper:       helper,
  questioner:   questioner,
  resources:    Resources{},
  cognitoActor: &actions.CognitoActions{CognitoClient:
 cognitoidentityprovider.NewFromConfig(sdkConfig)},
 }
 scenario.resources.init(scenario.cognitoActor, questioner)
 return scenario
}

// AddMigrateUserTrigger adds a Lambda handler as an invocation target for the
 MigrateUser trigger.
func (runner *MigrateUser) AddMigrateUserTrigger(ctx context.Context, userPoolId
 string, functionArn string) {
 log.Printf("Let's add a Lambda function to handle the MigrateUser trigger from
 Cognito.\n" +
  "This trigger happens when an unknown user signs in, and lets your function
 take action before Cognito\n" +
  "rejects the user.\n\n")
 err := runner.cognitoActor.UpdateTriggers(
  ctx, userPoolId,
  actions.TriggerInfo{Trigger: actions.UserMigration, HandlerArn:
 aws.String(functionArn)})
 if err != nil {
  panic(err)
 }
 log.Printf("Lambda function %v added to user pool %v to handle the MigrateUser
 trigger.\n",
  functionArn, userPoolId)

 log.Println(strings.Repeat("-", 88))
}
```

```
// SignInUser adds a new user to the known users table and signs that user in to
 Amazon Cognito.
func (runner *MigrateUser) SignInUser(ctx context.Context, usersTable string,
 clientId string) (bool, actions.User) {
 log.Println("Let's sign in a user to your Cognito user pool. When the username
 and email matches an entry in the\n" +
  "DynamoDB known users table, the email is automatically verified and the user
 is migrated to the Cognito user pool.")

 user := actions.User{}
 user.UserName = runner.questioner.Ask("\nEnter a username:")
 user.UserEmail = runner.questioner.Ask("\nEnter an email that you own. This
 email will be used to confirm user migration\n" +
  "during this example:")

 runner.helper.AddKnownUser(ctx, usersTable, user)

 var err error
 var resetRequired *types.PasswordResetRequiredException
 var authResult *types.AuthenticationResultType
 signedIn := false
 for !signedIn && resetRequired == nil {
  log.Printf("Signing in to Cognito as user '%v'. The expected result is a
 PasswordResetRequiredException.\n\n", user.UserName)
  authResult, err = runner.cognitoActor.SignIn(ctx, clientId, user.UserName, "_")
  if err != nil {
   if errors.As(err, &resetRequired) {
    log.Printf("\nUser '%v' is not in the Cognito user pool but was found in the
 DynamoDB known users table.\n"+
     "User migration is started and a password reset is required.",
 user.UserName)
   } else {
    panic(err)
   }
  } else {
   log.Printf("User '%v' successfully signed in. This is unexpected and probably
 means you have not\n"+
    "cleaned up a previous run of this scenario, so the user exist in the Cognito
 user pool.\n"+
    "You can continue this example and select to clean up resources, or manually
 remove\n"+
    "the user from your user pool and try again.", user.UserName)
```

```
    runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
  *authResult.AccessToken)
    signedIn = true
   }
  }

  log.Println(strings.Repeat("-", 88))
  return resetRequired != nil, user
}

// ResetPassword starts a password recovery flow.
func (runner *MigrateUser) ResetPassword(ctx context.Context, clientId string,
  user actions.User) {
  wantCode := runner.questioner.AskBool(fmt.Sprintf("In order to migrate the user
  to Cognito, you must be able to receive a confirmation\n"+
   "code by email at %v. Do you want to send a code (y/n)?", user.UserEmail), "y")
  if !wantCode {
   log.Println("To complete this example and successfully migrate a user to
  Cognito, you must enter an email\n" +
    "you own that can receive a confirmation code.")
   return
  }
  codeDelivery, err := runner.cognitoActor.ForgotPassword(ctx, clientId,
  user.UserName)
  if err != nil {
   panic(err)
  }
  log.Printf("\nA confirmation code has been sent to %v.",
  *codeDelivery.Destination)
  code := runner.questioner.Ask("Check your email and enter it here:")

  confirmed := false
  password := runner.questioner.AskPassword("\nEnter a password that has at least
  eight characters, uppercase, lowercase, numbers and symbols.\n"+
   "(the password will not display as you type):", 8)
  for !confirmed {
   log.Printf("\nConfirming password reset for user '%v'.\n", user.UserName)
   err = runner.cognitoActor.ConfirmForgotPassword(ctx, clientId, code,
  user.UserName, password)
   if err != nil {
    var invalidPassword *types.InvalidPasswordException
    if errors.As(err, &invalidPassword) {
     password = runner.questioner.AskPassword("\nEnter another password:", 8)
    } else {
```

```
    panic(err)
   }
  } else {
   confirmed = true
  }
 }
 log.Printf("User '%v' successfully confirmed and migrated.\n", user.UserName)
 log.Println("Signing in with your username and password...")
 authResult, err := runner.cognitoActor.SignIn(ctx, clientId, user.UserName,
 password)
 if err != nil {
  panic(err)
 }
 log.Printf("Successfully signed in. Your access token starts with: %v...\n",
 (*authResult.AccessToken)[:10])
 runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
 *authResult.AccessToken)

 log.Println(strings.Repeat("-", 88))
}

// Run runs the scenario.
func (runner *MigrateUser) Run(ctx context.Context, stackName string) {
 defer func() {
  if r := recover(); r != nil {
   log.Println("Something went wrong with the demo.")
   runner.resources.Cleanup(ctx)
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Printf("Welcome\n")

 log.Println(strings.Repeat("-", 88))

 stackOutputs, err := runner.helper.GetStackOutputs(ctx, stackName)
 if err != nil {
  panic(err)
 }
 runner.resources.userPoolId = stackOutputs["UserPoolId"]

 runner.AddMigrateUserTrigger(ctx, stackOutputs["UserPoolId"],
 stackOutputs["MigrateUserFunctionArn"])
```

```
 runner.resources.triggers = append(runner.resources.triggers,
 actions.UserMigration)
 resetNeeded, user := runner.SignInUser(ctx, stackOutputs["TableName"],
 stackOutputs["UserPoolClientId"])
 if resetNeeded {
  runner.helper.ListRecentLogEvents(ctx, stackOutputs["MigrateUserFunction"])
  runner.ResetPassword(ctx, stackOutputs["UserPoolClientId"], user)
 }

 runner.resources.Cleanup(ctx)

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

Gérez le déclencheur `MigrateUser` avec une fonction Lambda.

```
import (
 "context"
 "log"
 "os"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
)

const TABLE_NAME = "TABLE_NAME"

// UserInfo defines structured user data that can be marshalled to a DynamoDB
 format.
type UserInfo struct {
 UserName  string `dynamodbav:"UserName"`
 UserEmail string `dynamodbav:"UserEmail"`
}
```

```
type handler struct {
 dynamoClient *dynamodb.Client
}

// HandleRequest handles the MigrateUser event by looking up a user in an Amazon
 DynamoDB table and
// specifying whether they should be migrated to the user pool.
func (h *handler) HandleRequest(ctx context.Context, event
 events.CognitoEventUserPoolsMigrateUser)
 (events.CognitoEventUserPoolsMigrateUser, error) {
 log.Printf("Received migrate trigger from %v for user '%v'",
 event.TriggerSource, event.UserName)
 if event.TriggerSource != "UserMigration_Authentication" {
  return event, nil
 }
 tableName := os.Getenv(TABLE_NAME)
 user := UserInfo{
  UserName: event.UserName,
 }
 log.Printf("Looking up user '%v' in table %v.\n", user.UserName, tableName)
 filterEx := expression.Name("UserName").Equal(expression.Value(user.UserName))
 expr, err := expression.NewBuilder().WithFilter(filterEx).Build()
 if err != nil {
  log.Printf("Error building expression to query for user '%v'.\n",
 user.UserName)
  return event, err
 }
 output, err := h.dynamoClient.Scan(ctx, &dynamodb.ScanInput{
  TableName:                 aws.String(tableName),
  FilterExpression:          expr.Filter(),
  ExpressionAttributeNames:  expr.Names(),
  ExpressionAttributeValues: expr.Values(),
 })
 if err != nil {
  log.Printf("Error looking up user '%v'.\n", user.UserName)
  return event, err
 }
 if len(output.Items) == 0 {
  log.Printf("User '%v' not found, not migrating user.\n", user.UserName)
  return event, err
 }

 var users []UserInfo
```

```
 err = attributevalue.UnmarshalListOfMaps(output.Items, &users)
 if err != nil {
  log.Printf("Couldn't unmarshal DynamoDB items. Here's why: %v\n", err)
  return event, err
 }

 user = users[0]
 log.Printf("UserName '%v' found with email %v. User is migrated and must reset
 password.\n", user.UserName, user.UserEmail)
 event.CognitoEventUserPoolsMigrateUserResponse.UserAttributes =
 map[string]string{
  "email":          user.UserEmail,
  "email_verified": "true", // email_verified is required for the forgot password
 flow.
 }
 event.CognitoEventUserPoolsMigrateUserResponse.FinalUserStatus =
 "RESET_REQUIRED"
 event.CognitoEventUserPoolsMigrateUserResponse.MessageAction = "SUPPRESS"

 return event, err
}

func main() {
 ctx := context.Background()
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Panicln(err)
 }
 h := handler{
  dynamoClient: dynamodb.NewFromConfig(sdkConfig),
 }
 lambda.Start(h.HandleRequest)
}
```

Créez une structure qui exécute les tâches courantes.

```
import (
 "context"
 "log"
 "strings"
```

```
 "time"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudformation"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// IScenarioHelper defines common functions used by the workflows in this
 example.
type IScenarioHelper interface {
 Pause(secs int)
 GetStackOutputs(ctx context.Context, stackName string) (actions.StackOutputs,
 error)
 PopulateUserTable(ctx context.Context, tableName string)
 GetKnownUsers(ctx context.Context, tableName string) (actions.UserList, error)
 AddKnownUser(ctx context.Context, tableName string, user actions.User)
 ListRecentLogEvents(ctx context.Context, functionName string)
}

// ScenarioHelper contains AWS wrapper structs used by the workflows in this
 example.
type ScenarioHelper struct {
 questioner  demotools.IQuestioner
 dynamoActor *actions.DynamoActions
 cfnActor    *actions.CloudFormationActions
 cwlActor    *actions.CloudWatchLogsActions
 isTestRun   bool
}

// NewScenarioHelper constructs a new scenario helper.
func NewScenarioHelper(sdkConfig aws.Config, questioner demotools.IQuestioner)
 ScenarioHelper {
 scenario := ScenarioHelper{
  questioner:  questioner,
  dynamoActor: &actions.DynamoActions{DynamoClient:
 dynamodb.NewFromConfig(sdkConfig)},
  cfnActor:    &actions.CloudFormationActions{CfnClient:
 cloudformation.NewFromConfig(sdkConfig)},
  cwlActor:    &actions.CloudWatchLogsActions{CwlClient:
 cloudwatchlogs.NewFromConfig(sdkConfig)},
 }
```

```
 return scenario
}

// Pause waits for the specified number of seconds.
func (helper ScenarioHelper) Pause(secs int) {
 if !helper.isTestRun {
  time.Sleep(time.Duration(secs) * time.Second)
 }
}

// GetStackOutputs gets the outputs from the specified CloudFormation stack in a
 structured format.
func (helper ScenarioHelper) GetStackOutputs(ctx context.Context, stackName
 string) (actions.StackOutputs, error) {
 return helper.cfnActor.GetOutputs(ctx, stackName), nil
}

// PopulateUserTable fills the known user table with example data.
func (helper ScenarioHelper) PopulateUserTable(ctx context.Context, tableName
 string) {
 log.Printf("First, let's add some users to the DynamoDB %v table we'll use for
 this example.\n", tableName)
 err := helper.dynamoActor.PopulateTable(ctx, tableName)
 if err != nil {
  panic(err)
 }
}

// GetKnownUsers gets the users from the known users table in a structured
 format.
func (helper ScenarioHelper) GetKnownUsers(ctx context.Context, tableName string)
 (actions.UserList, error) {
 knownUsers, err := helper.dynamoActor.Scan(ctx, tableName)
 if err != nil {
  log.Printf("Couldn't get known users from table %v. Here's why: %v\n",
 tableName, err)
 }
 return knownUsers, err
}

// AddKnownUser adds a user to the known users table.
func (helper ScenarioHelper) AddKnownUser(ctx context.Context, tableName string,
 user actions.User) {
```

```
 log.Printf("Adding user '%v' with email '%v' to the DynamoDB known users
 table...\n",
  user.UserName, user.UserEmail)
 err := helper.dynamoActor.AddUser(ctx, tableName, user)
 if err != nil {
  panic(err)
 }
}

// ListRecentLogEvents gets the most recent log stream and events for the
 specified Lambda function and displays them.
func (helper ScenarioHelper) ListRecentLogEvents(ctx context.Context,
 functionName string) {
 log.Println("Waiting a few seconds to let Lambda write to CloudWatch Logs...")
 helper.Pause(10)
 log.Println("Okay, let's check the logs to find what's happened recently with
 your Lambda function.")
 logStream, err := helper.cwlActor.GetLatestLogStream(ctx, functionName)
 if err != nil {
  panic(err)
 }
 log.Printf("Getting some recent events from log stream %v\n",
 *logStream.LogStreamName)
 events, err := helper.cwlActor.GetLogEvents(ctx, functionName,
 *logStream.LogStreamName, 10)
 if err != nil {
  panic(err)
 }
 for _, event := range events {
  log.Printf("\t%v", *event.Message)
 }
 log.Println(strings.Repeat("-", 88))
}
```

Créez une structure qui encapsule les actions Amazon Cognito.

```
import (
 "context"
 "errors"
 "log"
```

```
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
)

type CognitoActions struct {
 CognitoClient *cognitoidentityprovider.Client
}



// Trigger and TriggerInfo define typed data for updating an Amazon Cognito
 trigger.
type Trigger int

const (
 PreSignUp Trigger = iota
 UserMigration
 PostAuthentication
)

type TriggerInfo struct {
 Trigger    Trigger
 HandlerArn *string
}

// UpdateTriggers adds or removes Lambda triggers for a user pool. When a trigger
 is specified with a `nil` value,
// it is removed from the user pool.
func (actor CognitoActions) UpdateTriggers(ctx context.Context, userPoolId
 string, triggers ...TriggerInfo) error {
 output, err := actor.CognitoClient.DescribeUserPool(ctx,
 &cognitoidentityprovider.DescribeUserPoolInput{
  UserPoolId: aws.String(userPoolId),
 })
 if err != nil {
  log.Printf("Couldn't get info about user pool %v. Here's why: %v\n",
 userPoolId, err)
  return err
 }
 lambdaConfig := output.UserPool.LambdaConfig
 for _, trigger := range triggers {
  switch trigger.Trigger {
```

```
  case PreSignUp:
   lambdaConfig.PreSignUp = trigger.HandlerArn
  case UserMigration:
   lambdaConfig.UserMigration = trigger.HandlerArn
  case PostAuthentication:
   lambdaConfig.PostAuthentication = trigger.HandlerArn
  }
 }
 _, err = actor.CognitoClient.UpdateUserPool(ctx,
 &cognitoidentityprovider.UpdateUserPoolInput{
  UserPoolId:   aws.String(userPoolId),
  LambdaConfig: lambdaConfig,
 })
 if err != nil {
  log.Printf("Couldn't update user pool %v. Here's why: %v\n", userPoolId, err)
 }
 return err
}



// SignUp signs up a user with Amazon Cognito.
func (actor CognitoActions) SignUp(ctx context.Context, clientId string, userName
 string, password string, userEmail string) (bool, error) {
 confirmed := false
 output, err := actor.CognitoClient.SignUp(ctx,
 &cognitoidentityprovider.SignUpInput{
  ClientId: aws.String(clientId),
  Password: aws.String(password),
  Username: aws.String(userName),
  UserAttributes: []types.AttributeType{
   {Name: aws.String("email"), Value: aws.String(userEmail)},
  },
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't sign up user %v. Here's why: %v\n", userName, err)
  }
 } else {
  confirmed = output.UserConfirmed
 }
```

```
 return confirmed, err
}



// SignIn signs in a user to Amazon Cognito using a username and password
 authentication flow.
func (actor CognitoActions) SignIn(ctx context.Context, clientId string, userName
 string, password string) (*types.AuthenticationResultType, error) {
 var authResult *types.AuthenticationResultType
 output, err := actor.CognitoClient.InitiateAuth(ctx,
 &cognitoidentityprovider.InitiateAuthInput{
  AuthFlow:       "USER_PASSWORD_AUTH",
  ClientId:       aws.String(clientId),
  AuthParameters: map[string]string{"USERNAME": userName, "PASSWORD": password},
 })
 if err != nil {
  var resetRequired *types.PasswordResetRequiredException
  if errors.As(err, &resetRequired) {
   log.Println(*resetRequired.Message)
  } else {
   log.Printf("Couldn't sign in user %v. Here's why: %v\n", userName, err)
  }
 } else {
  authResult = output.AuthenticationResult
 }
 return authResult, err
}



// ForgotPassword starts a password recovery flow for a user. This flow typically
 sends a confirmation code
// to the user's configured notification destination, such as email.
func (actor CognitoActions) ForgotPassword(ctx context.Context, clientId string,
 userName string) (*types.CodeDeliveryDetailsType, error) {
 output, err := actor.CognitoClient.ForgotPassword(ctx,
 &cognitoidentityprovider.ForgotPasswordInput{
  ClientId: aws.String(clientId),
  Username: aws.String(userName),
 })
 if err != nil {
  log.Printf("Couldn't start password reset for user '%v'. Here;s why: %v\n",
 userName, err)
```

```
 }
 return output.CodeDeliveryDetails, err
}



// ConfirmForgotPassword confirms a user with a confirmation code and a new
 password.
func (actor CognitoActions) ConfirmForgotPassword(ctx context.Context, clientId
 string, code string, userName string, password string) error {
 _, err := actor.CognitoClient.ConfirmForgotPassword(ctx,
 &cognitoidentityprovider.ConfirmForgotPasswordInput{
  ClientId:         aws.String(clientId),
  ConfirmationCode: aws.String(code),
  Password:         aws.String(password),
  Username:         aws.String(userName),
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't confirm user %v. Here's why: %v", userName, err)
  }
 }
 return err
}



// DeleteUser removes a user from the user pool.
func (actor CognitoActions) DeleteUser(ctx context.Context, userAccessToken
 string) error {
 _, err := actor.CognitoClient.DeleteUser(ctx,
 &cognitoidentityprovider.DeleteUserInput{
  AccessToken: aws.String(userAccessToken),
 })
 if err != nil {
  log.Printf("Couldn't delete user. Here's why: %v\n", err)
 }
 return err
}
```

```
// AdminCreateUser uses administrator credentials to add a user to a user pool.
 This method leaves the user
// in a state that requires they enter a new password next time they sign in.
func (actor CognitoActions) AdminCreateUser(ctx context.Context, userPoolId
 string, userName string, userEmail string) error {
 _, err := actor.CognitoClient.AdminCreateUser(ctx,
 &cognitoidentityprovider.AdminCreateUserInput{
  UserPoolId:     aws.String(userPoolId),
  Username:       aws.String(userName),
  MessageAction:  types.MessageActionTypeSuppress,
  UserAttributes: []types.AttributeType{{Name: aws.String("email"), Value:
 aws.String(userEmail)}},
 })
 if err != nil {
  var userExists *types.UsernameExistsException
  if errors.As(err, &userExists) {
   log.Printf("User %v already exists in the user pool.", userName)
   err = nil
  } else {
   log.Printf("Couldn't create user %v. Here's why: %v\n", userName, err)
  }
 }
 return err
}



// AdminSetUserPassword uses administrator credentials to set a password for a
 user without requiring a
// temporary password.
func (actor CognitoActions) AdminSetUserPassword(ctx context.Context, userPoolId
 string, userName string, password string) error {
 _, err := actor.CognitoClient.AdminSetUserPassword(ctx,
 &cognitoidentityprovider.AdminSetUserPasswordInput{
  Password:   aws.String(password),
  UserPoolId: aws.String(userPoolId),
  Username:   aws.String(userName),
  Permanent:  true,
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
```

```
  } else {
   log.Printf("Couldn't set password for user %v. Here's why: %v\n", userName,
 err)
  }
 }
 return err
}
```

Créez une structure qui encapsule les actions DynamoDB.

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// DynamoActions encapsulates the Amazon Simple Notification Service (Amazon SNS)
 actions
// used in the examples.
type DynamoActions struct {
 DynamoClient *dynamodb.Client
}

// User defines structured user data.
type User struct {
 UserName  string
 UserEmail string
 LastLogin *LoginInfo `dynamodbav:",omitempty"`
}

// LoginInfo defines structured custom login data.
type LoginInfo struct {
 UserPoolId string
 ClientId   string
 Time       string
```

```
}

// UserList defines a list of users.
type UserList struct {
 Users []User
}

// UserNameList returns the usernames contained in a UserList as a list of
 strings.
func (users *UserList) UserNameList() []string {
 names := make([]string, len(users.Users))
 for i := 0; i < len(users.Users); i++ {
  names[i] = users.Users[i].UserName
 }
 return names
}

// PopulateTable adds a set of test users to the table.
func (actor DynamoActions) PopulateTable(ctx context.Context, tableName string)
 error {
 var err error
 var item map[string]types.AttributeValue
 var writeReqs []types.WriteRequest
 for i := 1; i < 4; i++ {
  item, err = attributevalue.MarshalMap(User{UserName: fmt.Sprintf("test_user_
%v", i), UserEmail: fmt.Sprintf("test_email_%v@example.com", i)})
  if err != nil {
   log.Printf("Couldn't marshall user into DynamoDB format. Here's why: %v\n",
 err)
   return err
  }
  writeReqs = append(writeReqs, types.WriteRequest{PutRequest:
 &types.PutRequest{Item: item}})
 }
 _, err = actor.DynamoClient.BatchWriteItem(ctx, &dynamodb.BatchWriteItemInput{
  RequestItems: map[string][]types.WriteRequest{tableName: writeReqs},
 })
 if err != nil {
  log.Printf("Couldn't populate table %v with users. Here's why: %v\n",
 tableName, err)
 }
 return err
}
```

```
// Scan scans the table for all items.
func (actor DynamoActions) Scan(ctx context.Context, tableName string) (UserList,
 error) {
 var userList UserList
 output, err := actor.DynamoClient.Scan(ctx, &dynamodb.ScanInput{
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't scan table %v for items. Here's why: %v\n", tableName,
 err)
 } else {
  err = attributevalue.UnmarshalListOfMaps(output.Items, &userList.Users)
  if err != nil {
   log.Printf("Couldn't unmarshal items into users. Here's why: %v\n", err)
  }
 }
 return userList, err
}

// AddUser adds a user item to a table.
func (actor DynamoActions) AddUser(ctx context.Context, tableName string, user
 User) error {
 userItem, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshall user to item. Here's why: %v\n", err)
 }
 _, err = actor.DynamoClient.PutItem(ctx, &dynamodb.PutItemInput{
  Item:      userItem,
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't put item in table %v. Here's why: %v", tableName, err)
 }
 return err
}
```

Créez une structure qui englobe les actions CloudWatch Logs.

```
import (
 "context"
```

```
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs/types"
)

type CloudWatchLogsActions struct {
 CwlClient *cloudwatchlogs.Client
}

// GetLatestLogStream gets the most recent log stream for a Lambda function.
func (actor CloudWatchLogsActions) GetLatestLogStream(ctx context.Context,
 functionName string) (types.LogStream, error) {
 var logStream types.LogStream
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.DescribeLogStreams(ctx,
 &cloudwatchlogs.DescribeLogStreamsInput{
  Descending:   aws.Bool(true),
  Limit:        aws.Int32(1),
  LogGroupName: aws.String(logGroupName),
  OrderBy:      types.OrderByLastEventTime,
 })
 if err != nil {
  log.Printf("Couldn't get log streams for log group %v. Here's why: %v\n",
 logGroupName, err)
 } else {
  logStream = output.LogStreams[0]
 }
 return logStream, err
}

// GetLogEvents gets the most recent eventCount events from the specified log
 stream.
func (actor CloudWatchLogsActions) GetLogEvents(ctx context.Context, functionName
 string, logStreamName string, eventCount int32) (
 []types.OutputLogEvent, error) {
 var events []types.OutputLogEvent
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.GetLogEvents(ctx,
 &cloudwatchlogs.GetLogEventsInput{
  LogStreamName: aws.String(logStreamName),
  Limit:         aws.Int32(eventCount),
```

```
  LogGroupName:  aws.String(logGroupName),
 })
 if err != nil {
  log.Printf("Couldn't get log event for log stream %v. Here's why: %v\n",
 logStreamName, err)
 } else {
  events = output.Events
 }
 return events, err
}
```

Créez une structure qui englobe les actions. CloudFormation

```
import (
 "context"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudformation"
)

// StackOutputs defines a map of outputs from a specific stack.
type StackOutputs map[string]string

type CloudFormationActions struct {
 CfnClient *cloudformation.Client
}

// GetOutputs gets the outputs from a CloudFormation stack and puts them into a
 structured format.
func (actor CloudFormationActions) GetOutputs(ctx context.Context, stackName
 string) StackOutputs {
 output, err := actor.CfnClient.DescribeStacks(ctx,
 &cloudformation.DescribeStacksInput{
  StackName: aws.String(stackName),
 })
 if err != nil || len(output.Stacks) == 0 {
  log.Panicf("Couldn't find a CloudFormation stack named %v. Here's why: %v\n",
 stackName, err)
 }
```

```
 stackOutputs := StackOutputs{}
 for _, out := range output.Stacks[0].Outputs {
  stackOutputs[*out.OutputKey] = *out.OutputValue
 }
 return stackOutputs
}
```

Nettoyez les ressources.

```
import (
 "context"
 "log"
 "user_pools_and_lambda_triggers/actions"

 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// Resources keeps track of AWS resources created during an example and handles
// cleanup when the example finishes.
type Resources struct {
 userPoolId       string
 userAccessTokens []string
 triggers         []actions.Trigger

 cognitoActor *actions.CognitoActions
 questioner   demotools.IQuestioner
}

func (resources *Resources) init(cognitoActor *actions.CognitoActions, questioner
 demotools.IQuestioner) {
 resources.userAccessTokens = []string{}
 resources.triggers = []actions.Trigger{}
 resources.cognitoActor = cognitoActor
 resources.questioner = questioner
}

// Cleanup deletes all AWS resources created during an example.
func (resources *Resources) Cleanup(ctx context.Context) {
 defer func() {
  if r := recover(); r != nil {
```

```
    log.Printf("Something went wrong during cleanup.\n%v\n", r)
    log.Println("Use the AWS Management Console to remove any remaining resources
 \n" +
      "that were created for this scenario.")
   }
 }()

 wantDelete := resources.questioner.AskBool("Do you want to remove all of the AWS
 resources that were created "+
   "during this demo (y/n)?", "y")
 if wantDelete {
   for _, accessToken := range resources.userAccessTokens {
    err := resources.cognitoActor.DeleteUser(ctx, accessToken)
    if err != nil {
     log.Println("Couldn't delete user during cleanup.")
     panic(err)
    }
    log.Println("Deleted user.")
   }
   triggerList := make([]actions.TriggerInfo, len(resources.triggers))
   for i := 0; i < len(resources.triggers); i++ {
    triggerList[i] = actions.TriggerInfo{Trigger: resources.triggers[i],
 HandlerArn: nil}
   }
   err := resources.cognitoActor.UpdateTriggers(ctx, resources.userPoolId,
 triggerList...)
   if err != nil {
    log.Println("Couldn't update Cognito triggers during cleanup.")
    panic(err)
   }
   log.Println("Removed Cognito triggers from user pool.")
 } else {
   log.Println("Be sure to remove resources when you're done with them to avoid
 unexpected charges!")
 }
}
```

- Pour plus de détails sur l'API, consultez les rubriques suivantes dans la Référence des API du kit AWS SDK pour Go .
  - ConfirmForgotPassword

- DeleteUser
- ForgotPassword
- InitiateAuth
- SignUp
- UpdateUserPool

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Créer une API REST API Gateway pour suivre les données de la COVID-19

L'exemple de code suivant montre comment créer une API REST qui simule un système pour suivre les cas quotidiens de COVID-19 aux États-Unis, à l'aide de données fictives.

Python

Kit SDK for Python (Boto3)

Montre comment utiliser AWS Chalice avec le AWS SDK pour Python (Boto3) pour créer une API REST sans serveur qui utilise Amazon API Gateway et Amazon DynamoDB. AWS Lambda L'API REST simule un système qui suit les cas quotidiens de COVID-19 aux États-Unis à l'aide de données fictives. Découvrez comment :

- Utilisez AWS Chalice pour définir des routes dans les fonctions Lambda appelées pour gérer les requêtes REST qui passent par API Gateway.
- Utilisez les fonctions Lambda pour récupérer et stocker des données dans une table DynamoDB afin de répondre aux demandes REST.
- Définissez la structure des tables et les ressources des rôles de sécurité dans un AWS CloudFormation modèle.
- Utilisez AWS Chalice CloudFormation pour empaqueter et déployer toutes les ressources nécessaires.
- CloudFormation À utiliser pour nettoyer toutes les ressources créées.

Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Les services utilisés dans cet exemple

- API Gateway
- CloudFormation
- DynamoDB
- Lambda

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Créer une API REST de bibliothèque de prêt

L'exemple de code suivant montre comment créer une bibliothèque de prêt dans laquelle les clients peuvent emprunter et retourner des livres à l'aide d'une API REST soutenue par une base de données Amazon Aurora.

Python

Kit SDK for Python (Boto3)

Montre comment utiliser l' AWS SDK pour Python (Boto3) API Amazon Relational Database Service (Amazon RDS) et AWS Chalice pour créer une API REST soutenue par une base de données Amazon Aurora. Le service Web est entièrement sans serveur et représente une bibliothèque de prêt simple où les clients peuvent emprunter et retourner des livres. Découvrez comment :

- Créer et gérer un cluster de bases de données Aurora sans serveur.
- AWS Secrets Manager À utiliser pour gérer les informations d'identification de base de données.
- Implémenter une couche de stockage de données qui utilise Amazon RDS pour déplacer des données vers et hors de la base de données.
- Utilisez AWS Chalice pour déployer une API REST sans serveur sur Amazon API Gateway et. AWS Lambda
- Utiliser le package Requests (Requêtes) pour envoyer des requêtes au service web.

Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Les services utilisés dans cet exemple

- API Gateway
- Aurora
- Lambda
- Secrets Manager

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Créer une application de messagerie avec Step Functions

L'exemple de code suivant montre comment créer une application de AWS Step Functions messagerie qui extrait les enregistrements de messages d'une table de base de données.

Python

Kit SDK for Python (Boto3)

Montre comment utiliser le AWS SDK pour Python (Boto3) with AWS Step Functions pour créer une application de messagerie qui récupère les enregistrements de messages d'une table Amazon DynamoDB et les envoie via Amazon Simple Queue Service (Amazon SQS). La machine d'état intègre une AWS Lambda fonction permettant de scanner la base de données à la recherche de messages non envoyés.

- Créez une machine d’état qui extrait et met à jour des enregistrements de message d’une table Amazon DynamoDB.
- Mettez à jour la définition de la machine d’état pour envoyer des messages à Amazon Simple Queue Service (Amazon SQS).
- Démarrez et arrêtez les exécutions de la machine.
- Connectez-vous à Lambda, DynamoDB et Amazon SQS à partir d’une machine d’état à l’aide d’intégrations de services.

Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Les services utilisés dans cet exemple

- DynamoDB

- Lambda
- Amazon SQS
- Step Functions

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Création d'une application de gestion des ressources photographiques permettant aux utilisateurs de gérer les photos à l'aide d'étiquettes

Les exemples de code suivants montrent comment créer une application sans serveur permettant aux utilisateurs de gérer des photos à l'aide d'étiquettes.

.NET

SDK pour .NET

Montre comment développer une application de gestion de ressources photographiques qui détecte les étiquettes dans les images à l'aide d'Amazon Rekognition et les stocke pour les récupérer ultérieurement.

Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Pour explorer en profondeur l'origine de cet exemple, consultez l'article sur AWS Community.

Les services utilisés dans cet exemple

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

C++

SDK pour C++

Montre comment développer une application de gestion de ressources photographiques qui détecte les étiquettes dans les images à l'aide d'Amazon Rekognition et les stocke pour les récupérer ultérieurement.

Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Pour explorer en profondeur l'origine de cet exemple, consultez l'article sur AWS Community.

Les services utilisés dans cet exemple

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Java

SDK pour Java 2.x

Montre comment développer une application de gestion de ressources photographiques qui détecte les étiquettes dans les images à l'aide d'Amazon Rekognition et les stocke pour les récupérer ultérieurement.

Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Pour explorer en profondeur l'origine de cet exemple, consultez l'article sur AWS Community.

Les services utilisés dans cet exemple

- API Gateway
- DynamoDB
- Lambda

- Amazon Rekognition
- Amazon S3
- Amazon SNS

JavaScript

SDK pour JavaScript (v3)

Montre comment développer une application de gestion de ressources photographiques qui détecte les étiquettes dans les images à l'aide d'Amazon Rekognition et les stocke pour les récupérer ultérieurement.

Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Pour explorer en profondeur l'origine de cet exemple, consultez l'article sur AWS Community.

Les services utilisés dans cet exemple

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Kotlin

SDK pour Kotlin

Montre comment développer une application de gestion de ressources photographiques qui détecte les étiquettes dans les images à l'aide d'Amazon Rekognition et les stocke pour les récupérer ultérieurement.

Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Pour explorer en profondeur l'origine de cet exemple, consultez l'article sur AWS Community.

Les services utilisés dans cet exemple

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

PHP

Kit SDK pour PHP

Montre comment développer une application de gestion de ressources photographiques qui détecte les étiquettes dans les images à l'aide d'Amazon Rekognition et les stocke pour les récupérer ultérieurement.

Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Pour explorer en profondeur l'origine de cet exemple, consultez l'article sur AWS Community.

Les services utilisés dans cet exemple

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Rust

SDK pour Rust

Montre comment développer une application de gestion de ressources photographiques qui détecte les étiquettes dans les images à l'aide d'Amazon Rekognition et les stocke pour les récupérer ultérieurement.

Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Pour explorer en profondeur l'origine de cet exemple, consultez l'article sur AWS Community.

Les services utilisés dans cet exemple

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Créer une application de chat Websocket avec API Gateway

L'exemple de code suivant montre comment créer une application de chat desservie par une API Websocket basée sur Amazon API Gateway.

Python

Kit SDK for Python (Boto3)

Montre comment utiliser Amazon API Gateway V2 pour créer une API Websocket qui s'intègre à Amazon AWS Lambda DynamoDB. AWS SDK pour Python (Boto3)

- Créez une API WebSocket dans API Gateway.
- Définissez un gestionnaire Lambda qui stocke les connexions dans DynamoDB et publie des messages pour d'autres participants au chat.
- Connectez-vous à l'application de chat Websocket et envoyez des messages avec le package Websockets.

Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Les services utilisés dans cet exemple

- API Gateway
- DynamoDB
- Lambda

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Créez une application qui analyse les commentaires des clients et synthétise le son

Les exemples de code suivants montrent comment créer une application qui analyse les cartes de commentaires des clients, les traduit depuis leur langue d'origine, détermine leur sentiment et génère un fichier audio à partir du texte traduit.

.NET

SDK pour .NET

Cet exemple d'application analyse et stocke les cartes de commentaires des clients. Plus précisément, elle répond aux besoins d'un hôtel fictif situé à New York. L'hôtel reçoit les commentaires des clients dans différentes langues sous la forme de cartes de commentaires physiques. Ces commentaires sont chargés dans l'application via un client Web. Après avoir chargé l'image d'une carte de commentaires, les étapes suivantes se déroulent :

- Le texte est extrait de l'image à l'aide d'Amazon Textract.
- Amazon Comprehend détermine le sentiment du texte extrait et sa langue.
- Le texte extrait est traduit en anglais à l'aide d'Amazon Translate.
- Amazon Polly synthétise un fichier audio à partir du texte extrait.

L'application complète peut être déployée avec AWS CDK. Pour le code source et les instructions de déploiement, consultez le projet dans GitHub.

Les services utilisés dans cet exemple

- Amazon Comprehend
- Lambda

- Amazon Polly
- Amazon Textract
- Amazon Translate

Java

SDK pour Java 2.x

Cet exemple d'application analyse et stocke les cartes de commentaires des clients. Plus précisément, elle répond aux besoins d'un hôtel fictif situé à New York. L'hôtel reçoit les commentaires des clients dans différentes langues sous la forme de cartes de commentaires physiques. Ces commentaires sont chargés dans l'application via un client Web. Après avoir chargé l'image d'une carte de commentaires, les étapes suivantes se déroulent :

- Le texte est extrait de l'image à l'aide d'Amazon Textract.
- Amazon Comprehend détermine le sentiment du texte extrait et sa langue.
- Le texte extrait est traduit en anglais à l'aide d'Amazon Translate.
- Amazon Polly synthétise un fichier audio à partir du texte extrait.

L'application complète peut être déployée avec AWS CDK. Pour le code source et les instructions de déploiement, consultez le projet dans GitHub.

Les services utilisés dans cet exemple

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

JavaScript

SDK pour JavaScript (v3)

Cet exemple d'application analyse et stocke les cartes de commentaires des clients. Plus précisément, elle répond aux besoins d'un hôtel fictif situé à New York. L'hôtel reçoit les commentaires des clients dans différentes langues sous la forme de cartes de commentaires

physiques. Ces commentaires sont chargés dans l'application via un client Web. Après avoir chargé l'image d'une carte de commentaires, les étapes suivantes se déroulent :

- Le texte est extrait de l'image à l'aide d'Amazon Textract.
- Amazon Comprehend détermine le sentiment du texte extrait et sa langue.
- Le texte extrait est traduit en anglais à l'aide d'Amazon Translate.
- Amazon Polly synthétise un fichier audio à partir du texte extrait.

L'application complète peut être déployée avec AWS CDK. Pour le code source et les instructions de déploiement, consultez le projet dans GitHub. Les extraits suivants montrent comment le AWS SDK pour JavaScript est utilisé dans les fonctions Lambda.

```
import {
  ComprehendClient,
  DetectDominantLanguageCommand,
  DetectSentimentCommand,
} from "@aws-sdk/client-comprehend";

/**
 * Determine the language and sentiment of the extracted text.
 *
 * @param {{ source_text: string}} extractTextOutput
 */
export const handler = async (extractTextOutput) => {
  const comprehendClient = new ComprehendClient({});

  const detectDominantLanguageCommand = new DetectDominantLanguageCommand({
    Text: extractTextOutput.source_text,
  });

  // The source language is required for sentiment analysis and
  // translation in the next step.
  const { Languages } = await comprehendClient.send(
    detectDominantLanguageCommand,
  );

  const languageCode = Languages[0].LanguageCode;

  const detectSentimentCommand = new DetectSentimentCommand({
    Text: extractTextOutput.source_text,
    LanguageCode: languageCode,
  });
```

```
  const { Sentiment } = await comprehendClient.send(detectSentimentCommand);

  return {
    sentiment: Sentiment,
    language_code: languageCode,
  };
};
```

```
import {
  DetectDocumentTextCommand,
  TextractClient,
} from "@aws-sdk/client-textract";

/**
 * Fetch the S3 object from the event and analyze it using Amazon Textract.
 *
 * @param {import("@types/aws-lambda").EventBridgeEvent<"Object Created">}
 eventBridgeS3Event
 */
export const handler = async (eventBridgeS3Event) => {
  const textractClient = new TextractClient();

  const detectDocumentTextCommand = new DetectDocumentTextCommand({
    Document: {
      S3Object: {
        Bucket: eventBridgeS3Event.bucket,
        Name: eventBridgeS3Event.object,
      },
    },
  });

  // Textract returns a list of blocks. A block can be a line, a page, word, etc.
  // Each block also contains geometry of the detected text.
  // For more information on the Block type, see https://docs.aws.amazon.com/
textract/latest/dg/API_Block.html.
  const { Blocks } = await textractClient.send(detectDocumentTextCommand);

  // For the purpose of this example, we are only interested in words.
  const extractedWords = Blocks.filter((b) => b.BlockType === "WORD").map(
    (b) => b.Text,
  );
```

```
  return extractedWords.join(" ");
};
```

```
import { PollyClient, SynthesizeSpeechCommand } from "@aws-sdk/client-polly";
import { S3Client } from "@aws-sdk/client-s3";
import { Upload } from "@aws-sdk/lib-storage";

/**
 * Synthesize an audio file from text.
 *
 * @param {{ bucket: string, translated_text: string, object: string}}
 sourceDestinationConfig
 */
export const handler = async (sourceDestinationConfig) => {
  const pollyClient = new PollyClient({});

  const synthesizeSpeechCommand = new SynthesizeSpeechCommand({
    Engine: "neural",
    Text: sourceDestinationConfig.translated_text,
    VoiceId: "Ruth",
    OutputFormat: "mp3",
  });

  const { AudioStream } = await pollyClient.send(synthesizeSpeechCommand);

  const audioKey = `${sourceDestinationConfig.object}.mp3`;

  // Store the audio file in S3.
  const s3Client = new S3Client();
  const upload = new Upload({
    client: s3Client,
    params: {
      Bucket: sourceDestinationConfig.bucket,
      Key: audioKey,
      Body: AudioStream,
      ContentType: "audio/mp3",
    },
  });

  await upload.done();
  return audioKey;
};
```

```
import {
  TranslateClient,
  TranslateTextCommand,
} from "@aws-sdk/client-translate";

/**
 * Translate the extracted text to English.
 *
 * @param {{ extracted_text: string, source_language_code: string}}
 textAndSourceLanguage
 */
export const handler = async (textAndSourceLanguage) => {
  const translateClient = new TranslateClient({});

  const translateCommand = new TranslateTextCommand({
    SourceLanguageCode: textAndSourceLanguage.source_language_code,
    TargetLanguageCode: "en",
    Text: textAndSourceLanguage.extracted_text,
  });

  const { TranslatedText } = await translateClient.send(translateCommand);

  return { translated_text: TranslatedText };
};
```

Les services utilisés dans cet exemple

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

Ruby

Kit SDK pour Ruby

Cet exemple d'application analyse et stocke les cartes de commentaires des clients. Plus précisément, elle répond aux besoins d'un hôtel fictif situé à New York. L'hôtel reçoit les commentaires des clients dans différentes langues sous la forme de cartes de commentaires

physiques. Ces commentaires sont chargés dans l'application via un client Web. Après avoir chargé l'image d'une carte de commentaires, les étapes suivantes se déroulent :

- Le texte est extrait de l'image à l'aide d'Amazon Textract.
- Amazon Comprehend détermine le sentiment du texte extrait et sa langue.
- Le texte extrait est traduit en anglais à l'aide d'Amazon Translate.
- Amazon Polly synthétise un fichier audio à partir du texte extrait.

L'application complète peut être déployée avec AWS CDK. Pour le code source et les instructions de déploiement, consultez le projet dans GitHub.

Les services utilisés dans cet exemple

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Invoquer une fonction Lambda à partir d'un navigateur

L'exemple de code suivant montre comment invoquer une AWS Lambda fonction depuis un navigateur.

JavaScript

SDK pour JavaScript (v2)

Vous pouvez créer une application basée sur un navigateur qui utilise une AWS Lambda fonction pour mettre à jour une table Amazon DynamoDB avec les sélections des utilisateurs.

Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Les services utilisés dans cet exemple

- DynamoDB
- Lambda

SDK pour JavaScript (v3)

Vous pouvez créer une application basée sur un navigateur qui utilise une AWS Lambda fonction pour mettre à jour une table Amazon DynamoDB avec les sélections des utilisateurs. Cette application utilise la AWS SDK pour JavaScript version 3.

Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Les services utilisés dans cet exemple

- DynamoDB
- Lambda

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Transformation des données pour votre application avec S3 Object Lambda

L'exemple de code suivant illustre comment transformer des données pour votre application avec S3 Object Lambda.

.NET

SDK pour .NET

Explique comment ajouter du code personnalisé aux requêtes GET S3 standard afin de modifier l'objet demandé extrait de S3 pour qu'il réponde aux besoins du client ou de l'application qui le demande.

Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Les services utilisés dans cet exemple

- Lambda

- Amazon S3

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utiliser API Gateway pour invoquer une fonction Lambda

Les exemples de code suivants montrent comment créer une AWS Lambda fonction invoquée par Amazon API Gateway.

Java

SDK pour Java 2.x

Montre comment créer une AWS Lambda fonction à l'aide de l'API d'exécution Lambda Java. Cet exemple fait appel à différents AWS services pour réaliser un cas d'utilisation spécifique. Cet exemple montre comment créer une fonction Lambda invoquée par Amazon API Gateway qui analyse une table Amazon DynamoDB à la recherche d’anniversaires professionnels et utilise Amazon Simple Notification Service (Amazon SNS) pour envoyer un message texte à vos employés qui les félicitent à leur date d’anniversaire.

Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Les services utilisés dans cet exemple

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

JavaScript

SDK pour JavaScript (v3)

Montre comment créer une AWS Lambda fonction à l'aide de l'API JavaScript d'exécution Lambda. Cet exemple fait appel à différents AWS services pour réaliser un cas d'utilisation spécifique. Cet exemple montre comment créer une fonction Lambda invoquée par Amazon

API Gateway qui analyse une table Amazon DynamoDB à la recherche d'anniversaires professionnels et utilise Amazon Simple Notification Service (Amazon SNS) pour envoyer un message texte à vos employés qui les félicitent à leur date d'anniversaire.

Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Cet exemple est également disponible dans le AWS SDK pour JavaScript guide du développeur v3.

Les services utilisés dans cet exemple

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

Python

Kit SDK for Python (Boto3)

Cet exemple montre comment créer et utiliser une API REST Amazon API Gateway qui cible une fonction AWS Lambda . Le gestionnaire Lambda explique comment router en fonction des méthodes HTTP, comment obtenir des données à partir de la chaîne de requête, de l'en-tête et du corps, et comment renvoyer une réponse JSON.

- Déployer une fonction Lambda.
- Créer une API REST avec API Gateway.
- Créer une ressource REST qui cible la fonction Lambda.
- Accorder à API Gateway l'autorisation d'invoquer la fonction Lambda.
- Utiliser le package Requests (Requêtes) pour envoyer des requêtes à l'API REST.
- Nettoyer toutes les ressources créées lors de la démonstration.

Il est préférable de visionner cet exemple sur GitHub. Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Les services utilisés dans cet exemple

- API Gateway

- DynamoDB
- Lambda
- Amazon SNS

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utiliser les fonctions Step Functions pour invoquer des fonctions Lambda

L'exemple de code suivant montre comment créer une machine à AWS Step Functions états qui invoque des AWS Lambda fonctions en séquence.

Java

SDK pour Java 2.x

Montre comment créer un flux de travail AWS sans serveur en utilisant AWS Step Functions et le AWS SDK for Java 2.x. Chaque étape du flux de travail est implémentée à l'aide d'une AWS Lambda fonction.

Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Les services utilisés dans cet exemple

- DynamoDB
- Lambda
- Amazon SES
- Step Functions

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utilisent des événements planifiés pour invoquer une fonction Lambda

Les exemples de code suivants montrent comment créer une AWS Lambda fonction invoquée par un événement EventBridge planifié par Amazon.

Java

SDK pour Java 2.x

Montre comment créer un événement EventBridge planifié Amazon qui invoque une AWS Lambda fonction. Configurez EventBridge pour utiliser une expression cron afin de planifier le moment où la fonction Lambda est invoquée. Dans cet exemple, vous créez une fonction Lambda à l'aide de l'API d'exécution Lambda. Cet exemple fait appel à différents AWS services pour réaliser un cas d'utilisation spécifique. Cet exemple montre comment créer une application qui envoie un message texte mobile à vos employés pour les féliciter à leur date d'anniversaire.

Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Les services utilisés dans cet exemple

- CloudWatch Journaux
- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

JavaScript

SDK pour JavaScript (v3)

Montre comment créer un événement EventBridge planifié Amazon qui invoque une AWS Lambda fonction. Configurez EventBridge pour utiliser une expression cron afin de planifier le moment où la fonction Lambda est invoquée. Dans cet exemple, vous créez une fonction Lambda à l'aide de l'API d'exécution JavaScript Lambda. Cet exemple fait appel à différents AWS services pour réaliser un cas d'utilisation spécifique. Cet exemple montre comment créer une application qui envoie un message texte mobile à vos employés pour les féliciter à leur date d'anniversaire.

Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Cet exemple est également disponible dans le AWS SDK pour JavaScript guide du développeur v3.

Les services utilisés dans cet exemple

- CloudWatch Journaux
- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

Python

Kit SDK for Python (Boto3)

Cet exemple montre comment enregistrer une AWS Lambda fonction en tant que cible d'un EventBridge événement Amazon planifié. Le gestionnaire Lambda écrit un message convivial et les données complètes de l'événement dans Amazon CloudWatch Logs pour une récupération ultérieure.

- Déploie une fonction Lambda.
- Crée un événement EventBridge planifié et fait de la fonction Lambda la cible.
- Accorde l'autorisation de laisser EventBridge invoquer la fonction Lambda.
- Imprime les dernières données des CloudWatch journaux pour afficher le résultat des appels planifiés.
- Nettoie toutes les ressources créées lors de la démonstration.

Il est préférable de visionner cet exemple sur GitHub. Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Les services utilisés dans cet exemple

- CloudWatch Journaux
- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Utiliser l'API Amazon Neptune pour développer une fonction Lambda qui interroge les données d'un graphe

L'exemple de code suivant montre comment utiliser l'API Neptune pour interroger les données d'un graphe.

Java

SDK pour Java 2.x

Montre comment utiliser l'API Java Amazon Neptune pour créer une fonction Lambda qui interroge les données de graphe au sein du VPC.

Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Les services utilisés dans cet exemple

- Lambda
- Neptune

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Rédigez des données d'activité personnalisées à l'aide d'une fonction Lambda après l'authentification de l'utilisateur Amazon Cognito à l'aide d'un SDK AWS

L'exemple de code suivant illustre comment écrire des données d'activité personnalisées avec une fonction Lambda après l'authentification utilisateur Amazon Cognito.

- Utilisez les fonctions d'administrateur pour ajouter un utilisateur à un groupe d'utilisateurs.
- Configurez un groupe d'utilisateurs pour appeler une fonction Lambda pour le déclencheur `PostAuthentication`.

- Inscrivez le nouvel utilisateur dans Amazon Cognito.
- La fonction Lambda écrit des informations personnalisées dans des CloudWatch journaux et dans une table DynamoDB.
- Obtenez et affichez les données personnalisées à partir de la table DynamoDB, puis nettoyer les ressources.

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples de code AWS.

Exécutez un scénario interactif à une invite de commande.

```
import (
 "context"
 "errors"
 "log"
 "strings"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// ActivityLog separates the steps of this scenario into individual functions so
 that
// they are simpler to read and understand.
type ActivityLog struct {
 helper       IScenarioHelper
 questioner   demotools.IQuestioner
 resources    Resources
 cognitoActor *actions.CognitoActions
}
```

```
// NewActivityLog constructs a new activity log runner.
func NewActivityLog(sdkConfig aws.Config, questioner demotools.IQuestioner,
 helper IScenarioHelper) ActivityLog {
 scenario := ActivityLog{
  helper:       helper,
  questioner:   questioner,
  resources:    Resources{},
  cognitoActor: &actions.CognitoActions{CognitoClient:
 cognitoidentityprovider.NewFromConfig(sdkConfig)},
 }
 scenario.resources.init(scenario.cognitoActor, questioner)
 return scenario
}

// AddUserToPool selects a user from the known users table and uses administrator
 credentials to add the user to the user pool.
func (runner *ActivityLog) AddUserToPool(ctx context.Context, userPoolId string,
 tableName string) (string, string) {
 log.Println("To facilitate this example, let's add a user to the user pool using
 administrator privileges.")
 users, err := runner.helper.GetKnownUsers(ctx, tableName)
 if err != nil {
  panic(err)
 }
 user := users.Users[0]
 log.Printf("Adding known user %v to the user pool.\n", user.UserName)
 err = runner.cognitoActor.AdminCreateUser(ctx, userPoolId, user.UserName,
 user.UserEmail)
 if err != nil {
  panic(err)
 }
 pwSet := false
 password := runner.questioner.AskPassword("\nEnter a password that has at least
 eight characters, uppercase, lowercase, numbers and symbols.\n"+
  "(the password will not display as you type):", 8)
 for !pwSet {
  log.Printf("\nSetting password for user '%v'.\n", user.UserName)
  err = runner.cognitoActor.AdminSetUserPassword(ctx, userPoolId, user.UserName,
 password)
  if err != nil {
   var invalidPassword *types.InvalidPasswordException
   if errors.As(err, &invalidPassword) {
    password = runner.questioner.AskPassword("\nEnter another password:", 8)
```

```
   } else {
    panic(err)
   }
  } else {
   pwSet = true
  }
 }

 log.Println(strings.Repeat("-", 88))

 return user.UserName, password
}

// AddActivityLogTrigger adds a Lambda handler as an invocation target for the
 PostAuthentication trigger.
func (runner *ActivityLog) AddActivityLogTrigger(ctx context.Context, userPoolId
 string, activityLogArn string) {
 log.Println("Let's add a Lambda function to handle the PostAuthentication
 trigger from Cognito.\n" +
  "This trigger happens after a user is authenticated, and lets your function
 take action, such as logging\n" +
  "the outcome.")
 err := runner.cognitoActor.UpdateTriggers(
  ctx, userPoolId,
  actions.TriggerInfo{Trigger: actions.PostAuthentication, HandlerArn:
 aws.String(activityLogArn)})
 if err != nil {
  panic(err)
 }
 runner.resources.triggers = append(runner.resources.triggers,
 actions.PostAuthentication)
 log.Printf("Lambda function %v added to user pool %v to handle
 PostAuthentication Cognito trigger.\n",
  activityLogArn, userPoolId)

 log.Println(strings.Repeat("-", 88))
}

// SignInUser signs in as the specified user.
func (runner *ActivityLog) SignInUser(ctx context.Context, clientId string,
 userName string, password string) {
 log.Printf("Now we'll sign in user %v and check the results in the logs and the
 DynamoDB table.", userName)
 runner.questioner.Ask("Press Enter when you're ready.")
```

```
 authResult, err := runner.cognitoActor.SignIn(ctx, clientId, userName, password)
 if err != nil {
  panic(err)
 }
 log.Println("Sign in successful.",
  "The PostAuthentication Lambda handler writes custom information to CloudWatch
 Logs.")

 runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
 *authResult.AccessToken)
}

// GetKnownUserLastLogin gets the login info for a user from the Amazon DynamoDB
 table and displays it.
func (runner *ActivityLog) GetKnownUserLastLogin(ctx context.Context, tableName
 string, userName string) {
 log.Println("The PostAuthentication handler also writes login data to the
 DynamoDB table.")
 runner.questioner.Ask("Press Enter when you're ready to continue.")
 users, err := runner.helper.GetKnownUsers(ctx, tableName)
 if err != nil {
  panic(err)
 }
 for _, user := range users.Users {
  if user.UserName == userName {
   log.Println("The last login info for the user in the known users table is:")
   log.Printf("\t%+v", *user.LastLogin)
  }
 }
 log.Println(strings.Repeat("-", 88))
}

// Run runs the scenario.
func (runner *ActivityLog) Run(ctx context.Context, stackName string) {
 defer func() {
  if r := recover(); r != nil {
   log.Println("Something went wrong with the demo.")
   runner.resources.Cleanup(ctx)
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Printf("Welcome\n")
```

```
 log.Println(strings.Repeat("-", 88))

 stackOutputs, err := runner.helper.GetStackOutputs(ctx, stackName)
 if err != nil {
  panic(err)
 }
 runner.resources.userPoolId = stackOutputs["UserPoolId"]
 runner.helper.PopulateUserTable(ctx, stackOutputs["TableName"])
 userName, password := runner.AddUserToPool(ctx, stackOutputs["UserPoolId"],
 stackOutputs["TableName"])

 runner.AddActivityLogTrigger(ctx, stackOutputs["UserPoolId"],
 stackOutputs["ActivityLogFunctionArn"])
 runner.SignInUser(ctx, stackOutputs["UserPoolClientId"], userName, password)
 runner.helper.ListRecentLogEvents(ctx, stackOutputs["ActivityLogFunction"])
 runner.GetKnownUserLastLogin(ctx, stackOutputs["TableName"], userName)

 runner.resources.Cleanup(ctx)

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

Gérez le déclencheur `PostAuthentication` avec une fonction Lambda.

```
import (
 "context"
 "fmt"
 "log"
 "os"
 "time"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 dynamodbtypes "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
```

```
)

const TABLE_NAME = "TABLE_NAME"

// LoginInfo defines structured login data that can be marshalled to a DynamoDB
 format.
type LoginInfo struct {
 UserPoolId string `dynamodbav:"UserPoolId"`
 ClientId   string `dynamodbav:"ClientId"`
 Time       string `dynamodbav:"Time"`
}

// UserInfo defines structured user data that can be marshalled to a DynamoDB
 format.
type UserInfo struct {
 UserName  string    `dynamodbav:"UserName"`
 UserEmail string    `dynamodbav:"UserEmail"`
 LastLogin LoginInfo `dynamodbav:"LastLogin"`
}

// GetKey marshals the user email value to a DynamoDB key format.
func (user UserInfo) GetKey() map[string]dynamodbtypes.AttributeValue {
 userEmail, err := attributevalue.Marshal(user.UserEmail)
 if err != nil {
  panic(err)
 }
 return map[string]dynamodbtypes.AttributeValue{"UserEmail": userEmail}
}

type handler struct {
 dynamoClient *dynamodb.Client
}

// HandleRequest handles the PostAuthentication event by writing custom data to
 the logs and
// to an Amazon DynamoDB table.
func (h *handler) HandleRequest(ctx context.Context,
 event events.CognitoEventUserPoolsPostAuthentication)
 (events.CognitoEventUserPoolsPostAuthentication, error) {
 log.Printf("Received post authentication trigger from %v for user '%v'",
 event.TriggerSource, event.UserName)
 tableName := os.Getenv(TABLE_NAME)
 user := UserInfo{
  UserName:  event.UserName,
```

```
  UserEmail: event.Request.UserAttributes["email"],
  LastLogin: LoginInfo{
   UserPoolId: event.UserPoolID,
   ClientId:   event.CallerContext.ClientID,
   Time:       time.Now().Format(time.UnixDate),
  },
 }
 // Write to CloudWatch Logs.
 fmt.Printf("%#v", user)

 // Also write to an external system. This examples uses DynamoDB to demonstrate.
 userMap, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshal to DynamoDB map. Here's why: %v\n", err)
 } else if len(userMap) == 0 {
  log.Printf("User info marshaled to an empty map.")
 } else {
  _, err := h.dynamoClient.PutItem(ctx, &dynamodb.PutItemInput{
   Item:      userMap,
   TableName: aws.String(tableName),
  })
  if err != nil {
   log.Printf("Couldn't write to DynamoDB. Here's why: %v\n", err)
  } else {
   log.Printf("Wrote user info to DynamoDB table %v.\n", tableName)
  }
 }

 return event, nil
}

func main() {
 ctx := context.Background()
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Panicln(err)
 }
 h := handler{
  dynamoClient: dynamodb.NewFromConfig(sdkConfig),
 }
 lambda.Start(h.HandleRequest)
}
```

Créez une structure qui exécute les tâches courantes.

```
import (
 "context"
 "log"
 "strings"
 "time"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudformation"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// IScenarioHelper defines common functions used by the workflows in this
 example.
type IScenarioHelper interface {
 Pause(secs int)
 GetStackOutputs(ctx context.Context, stackName string) (actions.StackOutputs,
 error)
 PopulateUserTable(ctx context.Context, tableName string)
 GetKnownUsers(ctx context.Context, tableName string) (actions.UserList, error)
 AddKnownUser(ctx context.Context, tableName string, user actions.User)
 ListRecentLogEvents(ctx context.Context, functionName string)
}

// ScenarioHelper contains AWS wrapper structs used by the workflows in this
 example.
type ScenarioHelper struct {
 questioner  demotools.IQuestioner
 dynamoActor *actions.DynamoActions
 cfnActor    *actions.CloudFormationActions
 cwlActor    *actions.CloudWatchLogsActions
 isTestRun   bool
}

// NewScenarioHelper constructs a new scenario helper.
```

```
func NewScenarioHelper(sdkConfig aws.Config, questioner demotools.IQuestioner)
 ScenarioHelper {
 scenario := ScenarioHelper{
  questioner:  questioner,
  dynamoActor: &actions.DynamoActions{DynamoClient:
 dynamodb.NewFromConfig(sdkConfig)},
  cfnActor:    &actions.CloudFormationActions{CfnClient:
 cloudformation.NewFromConfig(sdkConfig)},
  cwlActor:    &actions.CloudWatchLogsActions{CwlClient:
 cloudwatchlogs.NewFromConfig(sdkConfig)},
 }
 return scenario
}

// Pause waits for the specified number of seconds.
func (helper ScenarioHelper) Pause(secs int) {
 if !helper.isTestRun {
  time.Sleep(time.Duration(secs) * time.Second)
 }
}

// GetStackOutputs gets the outputs from the specified CloudFormation stack in a
 structured format.
func (helper ScenarioHelper) GetStackOutputs(ctx context.Context, stackName
 string) (actions.StackOutputs, error) {
 return helper.cfnActor.GetOutputs(ctx, stackName), nil
}

// PopulateUserTable fills the known user table with example data.
func (helper ScenarioHelper) PopulateUserTable(ctx context.Context, tableName
 string) {
 log.Printf("First, let's add some users to the DynamoDB %v table we'll use for
 this example.\n", tableName)
 err := helper.dynamoActor.PopulateTable(ctx, tableName)
 if err != nil {
  panic(err)
 }
}

// GetKnownUsers gets the users from the known users table in a structured
 format.
func (helper ScenarioHelper) GetKnownUsers(ctx context.Context, tableName string)
 (actions.UserList, error) {
 knownUsers, err := helper.dynamoActor.Scan(ctx, tableName)
```

```
 if err != nil {
  log.Printf("Couldn't get known users from table %v. Here's why: %v\n",
 tableName, err)
 }
 return knownUsers, err
}

// AddKnownUser adds a user to the known users table.
func (helper ScenarioHelper) AddKnownUser(ctx context.Context, tableName string,
 user actions.User) {
 log.Printf("Adding user '%v' with email '%v' to the DynamoDB known users
 table...\n",
  user.UserName, user.UserEmail)
 err := helper.dynamoActor.AddUser(ctx, tableName, user)
 if err != nil {
  panic(err)
 }
}

// ListRecentLogEvents gets the most recent log stream and events for the
 specified Lambda function and displays them.
func (helper ScenarioHelper) ListRecentLogEvents(ctx context.Context,
 functionName string) {
 log.Println("Waiting a few seconds to let Lambda write to CloudWatch Logs...")
 helper.Pause(10)
 log.Println("Okay, let's check the logs to find what's happened recently with
 your Lambda function.")
 logStream, err := helper.cwlActor.GetLatestLogStream(ctx, functionName)
 if err != nil {
  panic(err)
 }
 log.Printf("Getting some recent events from log stream %v\n",
 *logStream.LogStreamName)
 events, err := helper.cwlActor.GetLogEvents(ctx, functionName,
 *logStream.LogStreamName, 10)
 if err != nil {
  panic(err)
 }
 for _, event := range events {
  log.Printf("\t%v", *event.Message)
 }
 log.Println(strings.Repeat("-", 88))
}
```

Créez une structure qui encapsule les actions Amazon Cognito.

```
import (
 "context"
 "errors"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
)

type CognitoActions struct {
 CognitoClient *cognitoidentityprovider.Client
}



// Trigger and TriggerInfo define typed data for updating an Amazon Cognito
 trigger.
type Trigger int

const (
 PreSignUp Trigger = iota
 UserMigration
 PostAuthentication
)

type TriggerInfo struct {
 Trigger    Trigger
 HandlerArn *string
}

// UpdateTriggers adds or removes Lambda triggers for a user pool. When a trigger
 is specified with a `nil` value,
// it is removed from the user pool.
func (actor CognitoActions) UpdateTriggers(ctx context.Context, userPoolId
 string, triggers ...TriggerInfo) error {
 output, err := actor.CognitoClient.DescribeUserPool(ctx,
 &cognitoidentityprovider.DescribeUserPoolInput{
```

```
  UserPoolId: aws.String(userPoolId),
 })
 if err != nil {
  log.Printf("Couldn't get info about user pool %v. Here's why: %v\n",
 userPoolId, err)
  return err
 }
 lambdaConfig := output.UserPool.LambdaConfig
 for _, trigger := range triggers {
  switch trigger.Trigger {
  case PreSignUp:
   lambdaConfig.PreSignUp = trigger.HandlerArn
  case UserMigration:
   lambdaConfig.UserMigration = trigger.HandlerArn
  case PostAuthentication:
   lambdaConfig.PostAuthentication = trigger.HandlerArn
  }
 }
 _, err = actor.CognitoClient.UpdateUserPool(ctx,
 &cognitoidentityprovider.UpdateUserPoolInput{
  UserPoolId:   aws.String(userPoolId),
  LambdaConfig: lambdaConfig,
 })
 if err != nil {
  log.Printf("Couldn't update user pool %v. Here's why: %v\n", userPoolId, err)
 }
 return err
}



// SignUp signs up a user with Amazon Cognito.
func (actor CognitoActions) SignUp(ctx context.Context, clientId string, userName
 string, password string, userEmail string) (bool, error) {
 confirmed := false
 output, err := actor.CognitoClient.SignUp(ctx,
 &cognitoidentityprovider.SignUpInput{
  ClientId: aws.String(clientId),
  Password: aws.String(password),
  Username: aws.String(userName),
  UserAttributes: []types.AttributeType{
   {Name: aws.String("email"), Value: aws.String(userEmail)},
  },
 })
```

```
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't sign up user %v. Here's why: %v\n", userName, err)
  }
 } else {
  confirmed = output.UserConfirmed
 }
 return confirmed, err
}



// SignIn signs in a user to Amazon Cognito using a username and password
 authentication flow.
func (actor CognitoActions) SignIn(ctx context.Context, clientId string, userName
 string, password string) (*types.AuthenticationResultType, error) {
 var authResult *types.AuthenticationResultType
 output, err := actor.CognitoClient.InitiateAuth(ctx,
 &cognitoidentityprovider.InitiateAuthInput{
  AuthFlow:       "USER_PASSWORD_AUTH",
  ClientId:       aws.String(clientId),
  AuthParameters: map[string]string{"USERNAME": userName, "PASSWORD": password},
 })
 if err != nil {
  var resetRequired *types.PasswordResetRequiredException
  if errors.As(err, &resetRequired) {
   log.Println(*resetRequired.Message)
  } else {
   log.Printf("Couldn't sign in user %v. Here's why: %v\n", userName, err)
  }
 } else {
  authResult = output.AuthenticationResult
 }
 return authResult, err
}



// ForgotPassword starts a password recovery flow for a user. This flow typically
 sends a confirmation code
// to the user's configured notification destination, such as email.
```

```
func (actor CognitoActions) ForgotPassword(ctx context.Context, clientId string,
 userName string) (*types.CodeDeliveryDetailsType, error) {
 output, err := actor.CognitoClient.ForgotPassword(ctx,
 &cognitoidentityprovider.ForgotPasswordInput{
  ClientId: aws.String(clientId),
  Username: aws.String(userName),
 })
 if err != nil {
  log.Printf("Couldn't start password reset for user '%v'. Here;s why: %v\n",
 userName, err)
 }
 return output.CodeDeliveryDetails, err
}



// ConfirmForgotPassword confirms a user with a confirmation code and a new
 password.
func (actor CognitoActions) ConfirmForgotPassword(ctx context.Context, clientId
 string, code string, userName string, password string) error {
 _, err := actor.CognitoClient.ConfirmForgotPassword(ctx,
 &cognitoidentityprovider.ConfirmForgotPasswordInput{
  ClientId:         aws.String(clientId),
  ConfirmationCode: aws.String(code),
  Password:         aws.String(password),
  Username:         aws.String(userName),
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't confirm user %v. Here's why: %v", userName, err)
  }
 }
 return err
}



// DeleteUser removes a user from the user pool.
func (actor CognitoActions) DeleteUser(ctx context.Context, userAccessToken
 string) error {
```

```
 _, err := actor.CognitoClient.DeleteUser(ctx,
 &cognitoidentityprovider.DeleteUserInput{
  AccessToken: aws.String(userAccessToken),
 })
 if err != nil {
  log.Printf("Couldn't delete user. Here's why: %v\n", err)
 }
 return err
}



// AdminCreateUser uses administrator credentials to add a user to a user pool.
 This method leaves the user
// in a state that requires they enter a new password next time they sign in.
func (actor CognitoActions) AdminCreateUser(ctx context.Context, userPoolId
 string, userName string, userEmail string) error {
 _, err := actor.CognitoClient.AdminCreateUser(ctx,
 &cognitoidentityprovider.AdminCreateUserInput{
  UserPoolId:     aws.String(userPoolId),
  Username:       aws.String(userName),
  MessageAction:  types.MessageActionTypeSuppress,
  UserAttributes: []types.AttributeType{{Name: aws.String("email"), Value:
 aws.String(userEmail)}},
 })
 if err != nil {
  var userExists *types.UsernameExistsException
  if errors.As(err, &userExists) {
   log.Printf("User %v already exists in the user pool.", userName)
   err = nil
  } else {
   log.Printf("Couldn't create user %v. Here's why: %v\n", userName, err)
  }
 }
 return err
}



// AdminSetUserPassword uses administrator credentials to set a password for a
 user without requiring a
// temporary password.
func (actor CognitoActions) AdminSetUserPassword(ctx context.Context, userPoolId
 string, userName string, password string) error {
```

```
 _, err := actor.CognitoClient.AdminSetUserPassword(ctx,
 &cognitoidentityprovider.AdminSetUserPasswordInput{
  Password:   aws.String(password),
  UserPoolId: aws.String(userPoolId),
  Username:   aws.String(userName),
  Permanent:  true,
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't set password for user %v. Here's why: %v\n", userName,
 err)
  }
 }
 return err
}
```

Créez une structure qui encapsule les actions DynamoDB.

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// DynamoActions encapsulates the Amazon Simple Notification Service (Amazon SNS)
 actions
// used in the examples.
type DynamoActions struct {
 DynamoClient *dynamodb.Client
}

// User defines structured user data.
```

```
type User struct {
 UserName  string
 UserEmail string
 LastLogin *LoginInfo `dynamodbav:",omitempty"`
}

// LoginInfo defines structured custom login data.
type LoginInfo struct {
 UserPoolId string
 ClientId   string
 Time       string
}

// UserList defines a list of users.
type UserList struct {
 Users []User
}

// UserNameList returns the usernames contained in a UserList as a list of
 strings.
func (users *UserList) UserNameList() []string {
 names := make([]string, len(users.Users))
 for i := 0; i < len(users.Users); i++ {
  names[i] = users.Users[i].UserName
 }
 return names
}

// PopulateTable adds a set of test users to the table.
func (actor DynamoActions) PopulateTable(ctx context.Context, tableName string)
 error {
 var err error
 var item map[string]types.AttributeValue
 var writeReqs []types.WriteRequest
 for i := 1; i < 4; i++ {
  item, err = attributevalue.MarshalMap(User{UserName: fmt.Sprintf("test_user_
%v", i), UserEmail: fmt.Sprintf("test_email_%v@example.com", i)})
  if err != nil {
   log.Printf("Couldn't marshall user into DynamoDB format. Here's why: %v\n",
 err)
   return err
  }
  writeReqs = append(writeReqs, types.WriteRequest{PutRequest:
 &types.PutRequest{Item: item}})
```

```
 }
 _, err = actor.DynamoClient.BatchWriteItem(ctx, &dynamodb.BatchWriteItemInput{
  RequestItems: map[string][]types.WriteRequest{tableName: writeReqs},
 })
 if err != nil {
  log.Printf("Couldn't populate table %v with users. Here's why: %v\n",
 tableName, err)
 }
 return err
}

// Scan scans the table for all items.
func (actor DynamoActions) Scan(ctx context.Context, tableName string) (UserList,
 error) {
 var userList UserList
 output, err := actor.DynamoClient.Scan(ctx, &dynamodb.ScanInput{
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't scan table %v for items. Here's why: %v\n", tableName,
 err)
 } else {
  err = attributevalue.UnmarshalListOfMaps(output.Items, &userList.Users)
  if err != nil {
   log.Printf("Couldn't unmarshal items into users. Here's why: %v\n", err)
  }
 }
 return userList, err
}

// AddUser adds a user item to a table.
func (actor DynamoActions) AddUser(ctx context.Context, tableName string, user
 User) error {
 userItem, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshall user to item. Here's why: %v\n", err)
 }
 _, err = actor.DynamoClient.PutItem(ctx, &dynamodb.PutItemInput{
  Item:      userItem,
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't put item in table %v. Here's why: %v", tableName, err)
 }
```

```
 return err
}
```

Créez une structure qui englobe les actions CloudWatch Logs.

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs/types"
)

type CloudWatchLogsActions struct {
 CwlClient *cloudwatchlogs.Client
}

// GetLatestLogStream gets the most recent log stream for a Lambda function.
func (actor CloudWatchLogsActions) GetLatestLogStream(ctx context.Context,
 functionName string) (types.LogStream, error) {
 var logStream types.LogStream
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.DescribeLogStreams(ctx,
 &cloudwatchlogs.DescribeLogStreamsInput{
  Descending:   aws.Bool(true),
  Limit:        aws.Int32(1),
  LogGroupName: aws.String(logGroupName),
  OrderBy:      types.OrderByLastEventTime,
 })
 if err != nil {
  log.Printf("Couldn't get log streams for log group %v. Here's why: %v\n",
 logGroupName, err)
 } else {
  logStream = output.LogStreams[0]
 }
 return logStream, err
}
```

```
// GetLogEvents gets the most recent eventCount events from the specified log
 stream.
func (actor CloudWatchLogsActions) GetLogEvents(ctx context.Context, functionName
 string, logStreamName string, eventCount int32) (
 []types.OutputLogEvent, error) {
 var events []types.OutputLogEvent
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.GetLogEvents(ctx,
 &cloudwatchlogs.GetLogEventsInput{
  LogStreamName: aws.String(logStreamName),
  Limit:         aws.Int32(eventCount),
  LogGroupName:  aws.String(logGroupName),
 })
 if err != nil {
  log.Printf("Couldn't get log event for log stream %v. Here's why: %v\n",
 logStreamName, err)
 } else {
  events = output.Events
 }
 return events, err
}
```

Créez une structure qui englobe les actions. CloudFormation

```
import (
 "context"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudformation"
)

// StackOutputs defines a map of outputs from a specific stack.
type StackOutputs map[string]string

type CloudFormationActions struct {
 CfnClient *cloudformation.Client
}
```

```
// GetOutputs gets the outputs from a CloudFormation stack and puts them into a
 structured format.
func (actor CloudFormationActions) GetOutputs(ctx context.Context, stackName
 string) StackOutputs {
 output, err := actor.CfnClient.DescribeStacks(ctx,
 &cloudformation.DescribeStacksInput{
  StackName: aws.String(stackName),
 })
 if err != nil || len(output.Stacks) == 0 {
  log.Panicf("Couldn't find a CloudFormation stack named %v. Here's why: %v\n",
 stackName, err)
 }
 stackOutputs := StackOutputs{}
 for _, out := range output.Stacks[0].Outputs {
  stackOutputs[*out.OutputKey] = *out.OutputValue
 }
 return stackOutputs
}
```

Nettoyez les ressources.

```
import (
 "context"
 "log"
 "user_pools_and_lambda_triggers/actions"

 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// Resources keeps track of AWS resources created during an example and handles
// cleanup when the example finishes.
type Resources struct {
 userPoolId       string
 userAccessTokens []string
 triggers         []actions.Trigger

 cognitoActor *actions.CognitoActions
 questioner   demotools.IQuestioner
}
```

```
func (resources *Resources) init(cognitoActor *actions.CognitoActions, questioner
 demotools.IQuestioner) {
 resources.userAccessTokens = []string{}
 resources.triggers = []actions.Trigger{}
 resources.cognitoActor = cognitoActor
 resources.questioner = questioner
}

// Cleanup deletes all AWS resources created during an example.
func (resources *Resources) Cleanup(ctx context.Context) {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong during cleanup.\n%v\n", r)
   log.Println("Use the AWS Management Console to remove any remaining resources
 \n" +
    "that were created for this scenario.")
  }
 }()

 wantDelete := resources.questioner.AskBool("Do you want to remove all of the AWS
 resources that were created "+
  "during this demo (y/n)?", "y")
 if wantDelete {
  for _, accessToken := range resources.userAccessTokens {
   err := resources.cognitoActor.DeleteUser(ctx, accessToken)
   if err != nil {
    log.Println("Couldn't delete user during cleanup.")
    panic(err)
   }
   log.Println("Deleted user.")
  }
  triggerList := make([]actions.TriggerInfo, len(resources.triggers))
  for i := 0; i < len(resources.triggers); i++ {
   triggerList[i] = actions.TriggerInfo{Trigger: resources.triggers[i],
 HandlerArn: nil}
  }
  err := resources.cognitoActor.UpdateTriggers(ctx, resources.userPoolId,
 triggerList...)
  if err != nil {
   log.Println("Couldn't update Cognito triggers during cleanup.")
   panic(err)
  }
  log.Println("Removed Cognito triggers from user pool.")
 } else {
```

```
  log.Println("Be sure to remove resources when you're done with them to avoid
 unexpected charges!")
 }
}
```

- Pour plus de détails sur l'API, consultez les rubriques suivantes dans la Référence des API du kit AWS SDK pour Go .
  - AdminCreateUser
  - AdminSetUserPassword
  - DeleteUser
  - InitiateAuth
  - UpdateUserPool

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Exemples sans serveur pour Lambda

Les exemples de code suivants montrent comment utiliser Lambda avec. AWS SDKs

Exemples

- Connexion à une base de données Amazon RDS dans une fonction Lambda
- Invoquer une fonction Lambda à partir d'un déclencheur Kinesis
- Invocation d'une fonction Lambda à partir d'un déclencheur DynamoDB
- Invocation d'une fonction Lambda à partir d'un déclencheur Amazon DocumentDB
- Invocation d'une fonction Lambda à partir d'un déclencheur Amazon MSK
- Invoquer une fonction Lambda à partir d'un déclencheur Amazon S3
- Invocation d'une fonction lambda à partir d'un déclencheur Amazon SNS
- Invoquer une fonction Lambda à partir d'un déclencheur Amazon SQS
- Signalement des échecs d'articles par lots pour les fonctions Lambda à l'aide d'un déclencheur Kinesis

- Signalement des échecs d'articles par lots pour les fonctions Lambda à l'aide d'un déclencheur DynamoDB
- Signalement des échecs d'articles par lots pour les fonctions Lambda à l'aide d'un déclencheur Amazon SQS

Connexion à une base de données Amazon RDS dans une fonction Lambda

Les exemples de code suivants illustrent comment implémenter une fonction Lambda qui se connecte à une base de données RDS. La fonction effectue une simple requête de base de données et renvoie le résultat.

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Connexion à une base de données Amazon RDS dans une fonction Lambda à l'aide de .NET.

```
using System.Data;
using System.Text.Json;
using Amazon.Lambda.APIGatewayEvents;
using Amazon.Lambda.Core;
using MySql.Data.MySqlClient;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace aws_rds;

public class InputModel
{
    public string key1 { get; set; }
```

```
    public string key2 { get; set; }
}

public class Function
{
    /// <summary>
    // Handles the Lambda function execution for connecting to RDS using IAM
 authentication.
    /// </summary>
    /// <param name="input">The input event data passed to the Lambda function</
param>
    /// <param name="context">The Lambda execution context that provides runtime
 information</param>
    /// <returns>A response object containing the execution result</returns>

    public async Task<APIGatewayProxyResponse>
 FunctionHandler(APIGatewayProxyRequest request, ILambdaContext context)
    {
        // Sample Input: {"body": "{\"key1\":\"20\", \"key2\":\"25\"}"}
        var input = JsonSerializer.Deserialize<InputModel>(request.Body);

        /// Obtain authentication token
        var authToken = RDSAuthTokenGenerator.GenerateAuthToken(
            Environment.GetEnvironmentVariable("RDS_ENDPOINT"),
            Convert.ToInt32(Environment.GetEnvironmentVariable("RDS_PORT")),
            Environment.GetEnvironmentVariable("RDS_USERNAME")
        );

        /// Build the Connection String with the Token
        string connectionString =
 $"Server={Environment.GetEnvironmentVariable("RDS_ENDPOINT")};" +

 $"Port={Environment.GetEnvironmentVariable("RDS_PORT")};" +

 $"Uid={Environment.GetEnvironmentVariable("RDS_USERNAME")};" +
                                  $"Pwd={authToken};";


        try
        {
            await using var connection = new MySqlConnection(connectionString);
            await connection.OpenAsync();

            const string sql = "SELECT @param1 + @param2 AS Sum";
```

```
            await using var command = new MySqlCommand(sql, connection);
            command.Parameters.AddWithValue("@param1", int.Parse(input.key1 ??
 "0"));
            command.Parameters.AddWithValue("@param2", int.Parse(input.key2 ??
 "0"));

            await using var reader = await command.ExecuteReaderAsync();
            if (await reader.ReadAsync())
            {
                int result = reader.GetInt32("Sum");

                //Sample Response: {"statusCode":200,"body":"{\"message\":\"The
 sum is: 45\"}","isBase64Encoded":false}
                return new APIGatewayProxyResponse
                {
                    StatusCode = 200,
                    Body = JsonSerializer.Serialize(new { message = $"The sum is:
 {result}" })
                };
            }

        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }

        return new APIGatewayProxyResponse
        {
            StatusCode = 500,
            Body = JsonSerializer.Serialize(new { error = "Internal server
 error" })
        };
    }
}
```

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Connexion à une base de données Amazon RDS dans une fonction Lambda à l'aide de Go.

```
/*
Golang v2 code here.
*/

package main

import (
 "context"
 "database/sql"
 "encoding/json"
 "fmt"
 "os"

 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/rds/auth"
 _ "github.com/go-sql-driver/mysql"
)

type MyEvent struct {
 Name string `json:"name"`
}

func HandleRequest(event *MyEvent) (map[string]interface{}, error) {

 var dbName string = os.Getenv("DatabaseName")
 var dbUser string = os.Getenv("DatabaseUser")
 var dbHost string = os.Getenv("DBHost") // Add hostname without https
 var dbPort int = os.Getenv("Port")      // Add port number
 var dbEndpoint string = fmt.Sprintf("%s:%d", dbHost, dbPort)
 var region string = os.Getenv("AWS_REGION")
```

```
 cfg, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  panic("configuration error: " + err.Error())
 }

 authenticationToken, err := auth.BuildAuthToken(
  context.TODO(), dbEndpoint, region, dbUser, cfg.Credentials)
 if err != nil {
  panic("failed to create authentication token: " + err.Error())
 }

 dsn := fmt.Sprintf("%s:%s@tcp(%s)/%s?tls=true&allowCleartextPasswords=true",
  dbUser, authenticationToken, dbEndpoint, dbName,
 )

 db, err := sql.Open("mysql", dsn)
 if err != nil {
  panic(err)
 }

 defer db.Close()

 var sum int
 err = db.QueryRow("SELECT ?+? AS sum", 3, 2).Scan(&sum)
 if err != nil {
  panic(err)
 }
 s := fmt.Sprint(sum)
 message := fmt.Sprintf("The selected sum is: %s", s)

 messageBytes, err := json.Marshal(message)
 if err != nil {
  return nil, err
 }

 messageString := string(messageBytes)
 return map[string]interface{}{
  "statusCode": 200,
  "headers":    map[string]string{"Content-Type": "application/json"},
  "body":       messageString,
 }, nil
}
```

```
func main() {
 lambda.Start(HandleRequest)
}
```

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Connexion à une base de données Amazon RDS dans une fonction Lambda à l'aide de Java.

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.rdsdata.RdsDataClient;
import software.amazon.awssdk.services.rdsdata.model.ExecuteStatementRequest;
import software.amazon.awssdk.services.rdsdata.model.ExecuteStatementResponse;
import software.amazon.awssdk.services.rdsdata.model.Field;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

public class RdsLambdaHandler implements
 RequestHandler<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent> {

    @Override
    public APIGatewayProxyResponseEvent handleRequest(APIGatewayProxyRequestEvent
 event, Context context) {
        APIGatewayProxyResponseEvent response = new
 APIGatewayProxyResponseEvent();
```

```
        try {
            // Obtain auth token
            String token = createAuthToken();

            // Define connection configuration
            String connectionString = String.format("jdbc:mysql://%s:%s/%s?
useSSL=true&requireSSL=true",
                    System.getenv("ProxyHostName"),
                    System.getenv("Port"),
                    System.getenv("DBName"));

            // Establish a connection to the database
            try (Connection connection =
 DriverManager.getConnection(connectionString, System.getenv("DBUserName"),
 token);
                 PreparedStatement statement =
 connection.prepareStatement("SELECT ? + ? AS sum")) {

                statement.setInt(1, 3);
                statement.setInt(2, 2);

                try (ResultSet resultSet = statement.executeQuery()) {
                    if (resultSet.next()) {
                        int sum = resultSet.getInt("sum");
                        response.setStatusCode(200);
                        response.setBody("The selected sum is: " + sum);
                    }
                }
            }

        } catch (Exception e) {
            response.setStatusCode(500);
            response.setBody("Error: " + e.getMessage());
        }

        return response;
    }

    private String createAuthToken() {
        // Create RDS Data Service client
        RdsDataClient rdsDataClient = RdsDataClient.builder()
                .region(Region.of(System.getenv("AWS_REGION")))
                .credentialsProvider(DefaultCredentialsProvider.create())
                .build();
```

```
        // Define authentication request
        ExecuteStatementRequest request = ExecuteStatementRequest.builder()
                .resourceArn(System.getenv("ProxyHostName"))
                .secretArn(System.getenv("DBUserName"))
                .database(System.getenv("DBName"))
                .sql("SELECT 'RDS IAM Authentication'")
                .build();

        // Execute request and obtain authentication token
        ExecuteStatementResponse response =
 rdsDataClient.executeStatement(request);
        Field tokenField = response.records().get(0).get(0);

        return tokenField.stringValue();
    }
}
```

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Connexion à une base de données Amazon RDS dans une fonction Lambda à l'aide de. JavaScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
/*
Node.js code here.
*/
// ES6+ example
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';
```

```
async function createAuthToken() {
  // Define connection authentication parameters
  const dbinfo = {

    hostname: process.env.ProxyHostName,
    port: process.env.Port,
    username: process.env.DBUserName,
    region: process.env.AWS_REGION,

  }

  // Create RDS Signer object
  const signer = new Signer(dbinfo);

  // Request authorization token from RDS, specifying the username
  const token = await signer.getAuthToken();
  return token;
}

async function dbOps() {

  // Obtain auth token
  const token = await createAuthToken();
  // Define connection configuration
  let connectionConfig = {
    host: process.env.ProxyHostName,
    user: process.env.DBUserName,
    password: token,
    database: process.env.DBName,
    ssl: 'Amazon RDS'
  }
  // Create the connection to the DB
  const conn = await mysql.createConnection(connectionConfig);
  // Obtain the result of the query
  const [res,] = await conn.execute('select ?+? as sum', [3, 2]);
  return res;

}

export const handler = async (event) => {
  // Execute database flow
  const result = await dbOps();
  // Return result
  return {
```

```
    statusCode: 200,
    body: JSON.stringify("The selected sum is: " + result[0].sum)
  }
};
```

Connexion à une base de données Amazon RDS dans une fonction Lambda à l'aide de. TypeScript

```
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

// RDS settings
// Using '!' (non-null assertion operator) to tell the TypeScript compiler that
 the DB settings are not null or undefined,
const proxy_host_name = process.env.PROXY_HOST_NAME!
const port = parseInt(process.env.PORT!)
const db_name = process.env.DB_NAME!
const db_user_name = process.env.DB_USER_NAME!
const aws_region = process.env.AWS_REGION!


async function createAuthToken(): Promise<string> {

    // Create RDS Signer object
    const signer = new Signer({
        hostname: proxy_host_name,
        port: port,
        region: aws_region,
        username: db_user_name
    });

    // Request authorization token from RDS, specifying the username
    const token = await signer.getAuthToken();
    return token;
}

async function dbOps(): Promise<mysql.QueryResult | undefined> {
    try {
        // Obtain auth token
        const token = await createAuthToken();
        const conn = await mysql.createConnection({
```

```
            host: proxy_host_name,
            user: db_user_name,
            password: token,
            database: db_name,
            ssl: 'Amazon RDS' // Ensure you have the CA bundle for SSL connection
        });
        const [rows, fields] = await conn.execute('SELECT ? + ? AS sum', [3, 2]);
        console.log('result:', rows);
        return rows;
    }
    catch (err) {
        console.log(err);
    }
}

export const lambdaHandler = async (event: any): Promise<{ statusCode: number;
 body: string }> => {
    // Execute database flow
    const result = await dbOps();

    // Return error is result is undefined
    if (result == undefined)
        return {
            statusCode: 500,
            body: JSON.stringify(`Error with connection to DB host`)
        }

    // Return result
    return {
        statusCode: 200,
        body: JSON.stringify(`The selected sum is: ${result[0].sum}`)
    };
};
```

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Connexion à une base de données Amazon RDS dans une fonction Lambda à l'aide de PHP.

```
<?php
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;
use Aws\Rds\AuthTokenGenerator;
use Aws\Credentials\CredentialProvider;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }


    private function getAuthToken(): string {
        // Define connection authentication parameters
        $dbConnection = [
            'hostname' => getenv('DB_HOSTNAME'),
            'port' => getenv('DB_PORT'),
            'username' => getenv('DB_USERNAME'),
            'region' => getenv('AWS_REGION'),
        ];
```

```
        // Create RDS AuthTokenGenerator object
        $generator = new
AuthTokenGenerator(CredentialProvider::defaultProvider());

        // Request authorization token from RDS, specifying the username
        return $generator->createToken(
            $dbConnection['hostname'] . ':' . $dbConnection['port'],
            $dbConnection['region'],
            $dbConnection['username']
        );
    }

    private function getQueryResults() {
        // Obtain auth token
        $token = $this->getAuthToken();

        // Define connection configuration
        $connectionConfig = [
            'host' => getenv('DB_HOSTNAME'),
            'user' => getenv('DB_USERNAME'),
            'password' => $token,
            'database' => getenv('DB_NAME'),
        ];

        // Create the connection to the DB
        $conn = new PDO(

"mysql:host={$connectionConfig['host']};dbname={$connectionConfig['database']}",
            $connectionConfig['user'],
            $connectionConfig['password'],
            [
                PDO::MYSQL_ATTR_SSL_CA => '/path/to/rds-ca-2019-root.pem',
                PDO::MYSQL_ATTR_SSL_VERIFY_SERVER_CERT => true,
            ]
        );

        // Obtain the result of the query
        $stmt = $conn->prepare('SELECT ?+? AS sum');
        $stmt->execute([3, 2]);

        return $stmt->fetch(PDO::FETCH_ASSOC);
    }
```

```
    /**
     * @param mixed $event
     * @param Context $context
     * @return array
     */
    public function handle(mixed $event, Context $context): array
    {
        $this->logger->info("Processing query");

        // Execute database flow
        $result = $this->getQueryResults();

        return [
            'sum' => $result['sum']
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Connexion à une base de données Amazon RDS dans une fonction Lambda à l'aide de Python.

```
import json
import os
import boto3
import pymysql

# RDS settings
proxy_host_name = os.environ['PROXY_HOST_NAME']
```

```
port = int(os.environ['PORT'])
db_name = os.environ['DB_NAME']
db_user_name = os.environ['DB_USER_NAME']
aws_region = os.environ['AWS_REGION']


# Fetch RDS Auth Token
def get_auth_token():
    client = boto3.client('rds')
    token = client.generate_db_auth_token(
        DBHostname=proxy_host_name,
        Port=port
        DBUsername=db_user_name
        Region=aws_region
    )
    return token

def lambda_handler(event, context):
    token = get_auth_token()
    try:
        connection = pymysql.connect(
            host=proxy_host_name,
            user=db_user_name,
            password=token,
            db=db_name,
            port=port,
            ssl={'ca': 'Amazon RDS'}  # Ensure you have the CA bundle for SSL
 connection
        )

        with connection.cursor() as cursor:
            cursor.execute('SELECT %s + %s AS sum', (3, 2))
            result = cursor.fetchone()

        return result

    except Exception as e:
        return (f"Error: {str(e)}")  # Return an error message if an exception
 occurs
```

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Connexion à une base de données Amazon RDS dans une fonction Lambda à l'aide de Ruby.

```
# Ruby code here.

require 'aws-sdk-rds'
require 'json'
require 'mysql2'

def lambda_handler(event:, context:)
  endpoint = ENV['DBEndpoint'] # Add the endpoint without https"
  port = ENV['Port']           # 3306
  user = ENV['DBUser']
  region = ENV['DBRegion']     # 'us-east-1'
  db_name = ENV['DBName']

  credentials = Aws::Credentials.new(
    ENV['AWS_ACCESS_KEY_ID'],
    ENV['AWS_SECRET_ACCESS_KEY'],
    ENV['AWS_SESSION_TOKEN']
  )
  rds_client = Aws::RDS::AuthTokenGenerator.new(
    region: region,
    credentials: credentials
  )

  token = rds_client.auth_token(
    endpoint: endpoint+ ':' + port,
    user_name: user,
    region: region
  )

  begin
    conn = Mysql2::Client.new(
```

```
      host: endpoint,
      username: user,
      password: token,
      port: port,
      database: db_name,
      sslca: '/var/task/global-bundle.pem',
      sslverify: true,
      enable_cleartext_plugin: true
    )
    a = 3
    b = 2
    result = conn.query("SELECT #{a} + #{b} AS sum").first['sum']
    puts result
    conn.close
    {
      statusCode: 200,
      body: result.to_json
    }
  rescue => e
    puts "Database connection failed due to #{e}"
  end
end
```

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Connexion à une base de données Amazon RDS dans une fonction Lambda à l'aide de Rust.

```
use aws_config::BehaviorVersion;
use aws_credential_types::provider::ProvideCredentials;
use aws_sigv4::{
    http_request::{sign, SignableBody, SignableRequest, SigningSettings},
    sign::v4,
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
```

```
use serde_json::{json, Value};
use sqlx::postgres::PgConnectOptions;
use std::env;
use std::time::{Duration, SystemTime};

const RDS_CERTS: &[u8] = include_bytes!("global-bundle.pem");

async fn generate_rds_iam_token(
    db_hostname: &str,
    port: u16,
    db_username: &str,
) -> Result<String, Error> {
    let config = aws_config::load_defaults(BehaviorVersion::v2024_03_28()).await;

    let credentials = config
        .credentials_provider()
        .expect("no credentials provider found")
        .provide_credentials()
        .await
        .expect("unable to load credentials");
    let identity = credentials.into();
    let region = config.region().unwrap().to_string();

    let mut signing_settings = SigningSettings::default();
    signing_settings.expires_in = Some(Duration::from_secs(900));
    signing_settings.signature_location =
 aws_sigv4::http_request::SignatureLocation::QueryParams;

    let signing_params = v4::SigningParams::builder()
        .identity(&identity)
        .region(&region)
        .name("rds-db")
        .time(SystemTime::now())
        .settings(signing_settings)
        .build()?;

    let url = format!(
        "https://{db_hostname}:{port}/?Action=connect&DBUser={db_user}",
        db_hostname = db_hostname,
        port = port,
        db_user = db_username
    );

    let signable_request =
```

```
        SignableRequest::new("GET", &url, std::iter::empty(),
 SignableBody::Bytes(&[]))
            .expect("signable request");

    let (signing_instructions, _signature) =
        sign(signable_request, &signing_params.into())?.into_parts();

    let mut url = url::Url::parse(&url).unwrap();
    for (name, value) in signing_instructions.params() {
        url.query_pairs_mut().append_pair(name, &value);
    }

    let response = url.to_string().split_off("https://".len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(handler)).await
}

async fn handler(_event: LambdaEvent<Value>) -> Result<Value, Error> {
    let db_host = env::var("DB_HOSTNAME").expect("DB_HOSTNAME must be set");
    let db_port = env::var("DB_PORT")
        .expect("DB_PORT must be set")
        .parse::<u16>()
        .expect("PORT must be a valid number");
    let db_name = env::var("DB_NAME").expect("DB_NAME must be set");
    let db_user_name = env::var("DB_USERNAME").expect("DB_USERNAME must be set");

    let token = generate_rds_iam_token(&db_host, db_port, &db_user_name).await?;

    let opts = PgConnectOptions::new()
        .host(&db_host)
        .port(db_port)
        .username(&db_user_name)
        .password(&token)
        .database(&db_name)
        .ssl_root_cert_from_pem(RDS_CERTS.to_vec())
        .ssl_mode(sqlx::postgres::PgSslMode::Require);

    let pool = sqlx::postgres::PgPoolOptions::new()
        .connect_with(opts)
```

```
        .await?;

    let result: i32 = sqlx::query_scalar("SELECT $1 + $2")
        .bind(3)
        .bind(2)
        .fetch_one(&pool)
        .await?;

    println!("Result: {:?}", result);

    Ok(json!({
        "statusCode": 200,
        "content-type": "text/plain",
        "body": format!("The selected sum is: {result}")
    }))
}
```

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Invoquer une fonction Lambda à partir d'un déclencheur Kinesis

Les exemples de code suivants montrent comment implémenter une fonction Lambda qui reçoit un événement déclenché par la réception d'enregistrements à partir d'un flux Kinesis. La fonction récupère la charge utile Kinesis, décode à partir de Base64 et enregistre le contenu de l'enregistrement.

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Kinesis avec Lambda à l'aide de .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegrationSampleCode;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task FunctionHandler(KinesisEvent evnt, ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return;
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                throw;
            }
        }
```

```
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}
```

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Kinesis avec Lambda à l'aide de Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent) error {
 if len(kinesisEvent.Records) == 0 {
  log.Printf("empty Kinesis event received")
  return nil
```

```
 }

 for _, record := range kinesisEvent.Records {
  log.Printf("processed Kinesis event with EventId: %v", record.EventID)
  recordDataBytes := record.Kinesis.Data
  recordDataText := string(recordDataBytes)
  log.Printf("record data: %v", recordDataText)
  // TODO: Do interesting work based on the new data
 }
 log.Printf("successfully processed %v records", len(kinesisEvent.Records))
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Kinesis avec Lambda à l'aide de Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;

public class Handler implements RequestHandler<KinesisEvent, Void> {
    @Override
    public Void handleRequest(final KinesisEvent event, final Context context) {
```

```
        LambdaLogger logger = context.getLogger();
        if (event.getRecords().isEmpty()) {
            logger.log("Empty Kinesis Event received");
            return null;
        }
        for (KinesisEvent.KinesisEventRecord record : event.getRecords()) {
            try {
                logger.log("Processed Event with EventId: "+record.getEventID());
                String data = new String(record.getKinesis().getData().array());
                logger.log("Data:"+ data);
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex) {
                logger.log("An error occurred:"+ex.getMessage());
                throw ex;
            }
        }
        logger.log("Successfully processed:"+event.getRecords().size()+"
 records");
        return null;
    }

}
```

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement Kinesis avec Lambda à l'aide de. JavaScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
```

```
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      throw err;
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

Utilisation d'un événement Kinesis avec Lambda à l'aide de. TypeScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    try {
```

```
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      throw err;
    }
    logger.info(`Successfully processed ${event.Records.length} records.`);
  }
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Kinesis avec Lambda à l'aide de PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Kinesis\KinesisHandler;
```

```
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends KinesisHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleKinesis(KinesisEvent $event, Context $context): void
    {
        $this->logger->info("Processing records");
        $records = $event->getRecords();
        foreach ($records as $record) {
            $data = $record->getData();
            $this->logger->info(json_encode($data));
            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Kinesis avec Lambda à l'aide de Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import base64
def lambda_handler(event, context):

    for record in event['Records']:
        try:
            print(f"Processed Kinesis Event - EventID: {record['eventID']}")
            record_data = base64.b64decode(record['kinesis']
['data']).decode('utf-8')
            print(f"Record Data: {record_data}")
            # TODO: Do interesting work based on the new data
        except Exception as e:
            print(f"An error occurred {e}")
            raise e
    print(f"Successfully processed {len(event['Records'])} records.")
```

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Kinesis avec Lambda à l'aide de Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue => err
      $stderr.puts "An error occurred #{err}"
      raise err
    end
  end
  puts "Successfully processed #{event['Records'].length} records."
end

def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('UTF-8')
  # Placeholder for actual async work
  # You can use Ruby's asynchronous programming tools like async/await or fibers
 here.
  return data
end
```

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Kinesis avec Lambda à l'aide de Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::kinesis::KinesisEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) -> Result<(), Error>
 {
    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    event.payload.records.iter().for_each(|record| {
        tracing::info!("EventId:
 {}",record.event_id.as_deref().unwrap_or_default());

        let record_data = std::str::from_utf8(&record.kinesis.data);

        match record_data {
            Ok(data) => {
                // log the record data
                tracing::info!("Data: {}", data);
            }
            Err(e) => {
                tracing::error!("Error: {}", e);
            }
        }
    });

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
```

```
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Invocation d'une fonction Lambda à partir d'un déclencheur DynamoDB

Les exemples de code suivants illustrent comment implémenter une fonction Lambda qui reçoit un événement déclenché par la réception d'enregistrements à partir d'un flux DynamoDB. La fonction récupère les données utiles DynamoDB et journalise le contenu de l'enregistrement.

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement DynamoDB avec Lambda en utilisant .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri
```

```
namespace AWSLambda_DDB;

public class Function
{
    public void FunctionHandler(DynamoDBEvent dynamoEvent, ILambdaContext
 context)
    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");

        foreach (var record in dynamoEvent.Records)
        {
            context.Logger.LogInformation($"Event ID: {record.EventID}");
            context.Logger.LogInformation($"Event Name: {record.EventName}");

            context.Logger.LogInformation(JsonSerializer.Serialize(record));
        }

        context.Logger.LogInformation("Stream processing complete.");
    }
}
```

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement DynamoDB avec Lambda en utilisant Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
```

```
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-lambda-go/events"
 "fmt"
)

func HandleRequest(ctx context.Context, event events.DynamoDBEvent) (*string,
 error) {
 if len(event.Records) == 0 {
  return nil, fmt.Errorf("received empty event")
 }

 for _, record := range event.Records {
   LogDynamoDBRecord(record)
 }

 message := fmt.Sprintf("Records processed: %d", len(event.Records))
 return &message, nil
}

func main() {
 lambda.Start(HandleRequest)
}

func LogDynamoDBRecord(record events.DynamoDBEventRecord){
 fmt.Println(record.EventID)
 fmt.Println(record.EventName)
 fmt.Printf("%+v\n", record.Change)
}
```

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement DynamoDB avec Lambda en utilisant Java.

```
import com.amazonaws.services.lambda.runtime.Context;
```

```
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import
 com.amazonaws.services.lambda.runtime.events.DynamodbEvent.DynamodbStreamRecord;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class example implements RequestHandler<DynamodbEvent, Void> {

    private static final Gson GSON = new
 GsonBuilder().setPrettyPrinting().create();

    @Override
    public Void handleRequest(DynamodbEvent event, Context context) {
        System.out.println(GSON.toJson(event));
        event.getRecords().forEach(this::logDynamoDBRecord);
        return null;
    }

    private void logDynamoDBRecord(DynamodbStreamRecord record) {
        System.out.println(record.getEventID());
        System.out.println(record.getEventName());
        System.out.println("DynamoDB Record: " +
 GSON.toJson(record.getDynamodb()));
    }
}
```

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement DynamoDB avec Lambda en utilisant. JavaScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
```

```
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
};

const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

Consommation d'un événement DynamoDB avec Lambda en utilisant. TypeScript

```
export const handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
}
const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement DynamoDB avec Lambda en utilisant PHP.

```
<?php
```

```
# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\DynamoDb\DynamoDbHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends DynamoDbHandler
{
    private StderrLogger $logger;

    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleDynamoDb(DynamoDbEvent $event, Context $context): void
    {
        $this->logger->info("Processing DynamoDb table items");
        $records = $event->getRecords();

        foreach ($records as $record) {
            $eventName = $record->getEventName();
            $keys = $record->getKeys();
            $old = $record->getOldImage();
            $new = $record->getNewImage();

            $this->logger->info("Event Name:".$eventName."\n");
            $this->logger->info("Keys:". json_encode($keys)."\n");
            $this->logger->info("Old Image:". json_encode($old)."\n");
            $this->logger->info("New Image:". json_encode($new));

            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }
```

```
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords items");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement DynamoDB avec Lambda en utilisant Python.

```
import json

def lambda_handler(event, context):
    print(json.dumps(event, indent=2))

    for record in event['Records']:
        log_dynamodb_record(record)

def log_dynamodb_record(record):
    print(record['eventID'])
    print(record['eventName'])
    print(f"DynamoDB Record: {json.dumps(record['dynamodb'])}")
```

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement DynamoDB avec Lambda en utilisant Ruby.

```
def lambda_handler(event:, context:)
    return 'received empty event' if event['Records'].empty?

    event['Records'].each do |record|
      log_dynamodb_record(record)
    end

    "Records processed: #{event['Records'].length}"
  end

  def log_dynamodb_record(record)
    puts record['eventID']
    puts record['eventName']
    puts "DynamoDB Record: #{JSON.generate(record['dynamodb'])}"
  end
```

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement DynamoDB avec Lambda en utilisant Rust.

```
use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord},
   };


// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
//tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<Event>) ->Result<(), Error> {

    let records = &event.payload.records;
    tracing::info!("event payload: {:?}",records);
    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    for record in records{
        log_dynamo_dbrecord(record);
    }

    tracing::info!("Dynamo db records processed");

    // Prepare the response
    Ok(())

}

fn log_dynamo_dbrecord(record: &EventRecord)-> Result<(), Error>{
    tracing::info!("EventId: {}", record.event_id);
    tracing::info!("EventName: {}", record.event_name);
    tracing::info!("DynamoDB Record: {:?}", record.change );
    Ok(())
```

```
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())

}
```

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Invocation d'une fonction Lambda à partir d'un déclencheur Amazon DocumentDB

Les exemples de code suivants illustrent comment implémenter une fonction Lambda qui reçoit un événement déclenché par la réception d'enregistrements à partir d'un flux de modification DocumentDB. La fonction récupère les données utiles DocumentDB et journalise le contenu de l'enregistrement.

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon DocumentDB avec Lambda en utilisant .NET.

```
using Amazon.Lambda.Core;
using System.Text.Json;
using System;
using System.Collections.Generic;
using System.Text.Json.Serialization;
//Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace LambdaDocDb;

public class Function
{

     /// <summary>
    /// Lambda function entry point to process Amazon DocumentDB events.
    /// </summary>
    /// <param name="event">The Amazon DocumentDB event.</param>
    /// <param name="context">The Lambda context object.</param>
    /// <returns>A string to indicate successful processing.</returns>
    public string FunctionHandler(Event evnt, ILambdaContext context)
    {

        foreach (var record in evnt.Events)
        {
            ProcessDocumentDBEvent(record, context);
        }

        return "OK";
    }

     private void ProcessDocumentDBEvent(DocumentDBEventRecord record,
 ILambdaContext context)
    {

        var eventData = record.Event;
        var operationType = eventData.OperationType;
        var databaseName = eventData.Ns.Db;
        var collectionName = eventData.Ns.Coll;
        var fullDocument = JsonSerializer.Serialize(eventData.FullDocument, new
 JsonSerializerOptions { WriteIndented = true });
```

```
        context.Logger.LogLine($"Operation type: {operationType}");
        context.Logger.LogLine($"Database: {databaseName}");
        context.Logger.LogLine($"Collection: {collectionName}");
        context.Logger.LogLine($"Full document:\n{fullDocument}");
    }



    public class Event
    {
        [JsonPropertyName("eventSourceArn")]
        public string EventSourceArn { get; set; }

        [JsonPropertyName("events")]
        public List<DocumentDBEventRecord> Events { get; set; }

        [JsonPropertyName("eventSource")]
        public string EventSource { get; set; }
    }

    public class DocumentDBEventRecord
    {
        [JsonPropertyName("event")]
        public EventData Event { get; set; }
    }

    public class EventData
    {
        [JsonPropertyName("_id")]
        public IdData Id { get; set; }

        [JsonPropertyName("clusterTime")]
        public ClusterTime ClusterTime { get; set; }

        [JsonPropertyName("documentKey")]
        public DocumentKey DocumentKey { get; set; }

        [JsonPropertyName("fullDocument")]
        public Dictionary<string, object> FullDocument { get; set; }

        [JsonPropertyName("ns")]
        public Namespace Ns { get; set; }

        [JsonPropertyName("operationType")]
```

```
        public string OperationType { get; set; }
    }

    public class IdData
    {
        [JsonPropertyName("_data")]
        public string Data { get; set; }
    }

    public class ClusterTime
    {
        [JsonPropertyName("$timestamp")]
        public Timestamp Timestamp { get; set; }
    }

    public class Timestamp
    {
        [JsonPropertyName("t")]
        public long T { get; set; }

        [JsonPropertyName("i")]
        public int I { get; set; }
    }

    public class DocumentKey
    {
        [JsonPropertyName("_id")]
        public Id Id { get; set; }
    }

    public class Id
    {
        [JsonPropertyName("$oid")]
        public string Oid { get; set; }
    }

    public class Namespace
    {
        [JsonPropertyName("db")]
        public string Db { get; set; }

        [JsonPropertyName("coll")]
        public string Coll { get; set; }
    }
```

```
}
```

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon DocumentDB avec Lambda en utilisant Go.

```
package main

import (
 "context"
 "encoding/json"
 "fmt"

 "github.com/aws/aws-lambda-go/lambda"
)

type Event struct {
 Events []Record `json:"events"`
}

type Record struct {
 Event struct {
  OperationType string `json:"operationType"`
  NS            struct {
   DB   string `json:"db"`
   Coll string `json:"coll"`
  } `json:"ns"`
  FullDocument interface{} `json:"fullDocument"`
 } `json:"event"`
}

func main() {
```

```
 lambda.Start(handler)
}

func handler(ctx context.Context, event Event) (string, error) {
 fmt.Println("Loading function")
 for _, record := range event.Events {
  logDocumentDBEvent(record)
 }

 return "OK", nil
}

func logDocumentDBEvent(record Record) {
 fmt.Printf("Operation type: %s\n", record.Event.OperationType)
 fmt.Printf("db: %s\n", record.Event.NS.DB)
 fmt.Printf("collection: %s\n", record.Event.NS.Coll)
 docBytes, _ := json.MarshalIndent(record.Event.FullDocument, "", "  ")
 fmt.Printf("Full document: %s\n", string(docBytes))
}
```

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon DocumentDB avec Lambda en utilisant Java.

```
import java.util.List;
import java.util.Map;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

public class Example implements RequestHandler<Map<String, Object>, String> {

    @SuppressWarnings("unchecked")
```

```
    @Override
    public String handleRequest(Map<String, Object> event, Context context) {
        List<Map<String, Object>> events = (List<Map<String, Object>>)
 event.get("events");
        for (Map<String, Object> record : events) {
            Map<String, Object> eventData = (Map<String, Object>)
 record.get("event");
            processEventData(eventData);
        }

        return "OK";
    }

    @SuppressWarnings("unchecked")
    private void processEventData(Map<String, Object> eventData) {
        String operationType = (String) eventData.get("operationType");
        System.out.println("operationType: %s".formatted(operationType));

        Map<String, Object> ns = (Map<String, Object>) eventData.get("ns");

        String db = (String) ns.get("db");
        System.out.println("db: %s".formatted(db));
        String coll = (String) ns.get("coll");
        System.out.println("coll: %s".formatted(coll));

        Map<String, Object> fullDocument = (Map<String, Object>)
 eventData.get("fullDocument");
        System.out.println("fullDocument: %s".formatted(fullDocument));
    }

}
```

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon DocumentDB avec Lambda à l'aide de. JavaScript

```
console.log('Loading function');
exports.handler = async (event, context) => {
    event.events.forEach(record => {
        logDocumentDBEvent(record);
    });
    return 'OK';
};

const logDocumentDBEvent = (record) => {
    console.log('Operation type: ' + record.event.operationType);
    console.log('db: ' + record.event.ns.db);
    console.log('collection: ' + record.event.ns.coll);
    console.log('Full document:', JSON.stringify(record.event.fullDocument, null,
 2));
};
```

Utilisation d'un événement Amazon DocumentDB avec Lambda à l'aide de TypeScript

```
import { DocumentDBEventRecord, DocumentDBEventSubscriptionContext } from 'aws-
lambda';

console.log('Loading function');

export const handler = async (
  event: DocumentDBEventSubscriptionContext,
  context: any
): Promise<string> => {
  event.events.forEach((record: DocumentDBEventRecord) => {
    logDocumentDBEvent(record);
  });
  return 'OK';
};

const logDocumentDBEvent = (record: DocumentDBEventRecord): void => {
  console.log('Operation type: ' + record.event.operationType);
  console.log('db: ' + record.event.ns.db);
  console.log('collection: ' + record.event.ns.coll);
  console.log('Full document:', JSON.stringify(record.event.fullDocument, null,
 2));
```

```
};
```

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon DocumentDB avec Lambda en utilisant PHP.

```
<?php

require __DIR__.'/vendor/autoload.php';

use Bref\Context\Context;
use Bref\Event\Handler;

class DocumentDBEventHandler implements Handler
{
    public function handle($event, Context $context): string
    {

        $events = $event['events'] ?? [];
        foreach ($events as $record) {
            $this->logDocumentDBEvent($record['event']);
        }
        return 'OK';
    }

    private function logDocumentDBEvent($event): void
    {
        // Extract information from the event record

        $operationType = $event['operationType'] ?? 'Unknown';
        $db = $event['ns']['db'] ?? 'Unknown';
        $collection = $event['ns']['coll'] ?? 'Unknown';
        $fullDocument = $event['fullDocument'] ?? [];
```

```
        // Log the event details

        echo "Operation type: $operationType\n";
        echo "Database: $db\n";
        echo "Collection: $collection\n";
        echo "Full document: " . json_encode($fullDocument, JSON_PRETTY_PRINT) .
 "\n";
    }
}
return new DocumentDBEventHandler();
```

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon DocumentDB avec Lambda en utilisant Python.

```
import json

def lambda_handler(event, context):
    for record in event.get('events', []):
        log_document_db_event(record)
    return 'OK'

def log_document_db_event(record):
    event_data = record.get('event', {})
    operation_type = event_data.get('operationType', 'Unknown')
    db = event_data.get('ns', {}).get('db', 'Unknown')
    collection = event_data.get('ns', {}).get('coll', 'Unknown')
    full_document = event_data.get('fullDocument', {})

    print(f"Operation type: {operation_type}")
    print(f"db: {db}")
    print(f"collection: {collection}")
```

```
    print("Full document:", json.dumps(full_document, indent=2))
```

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon DocumentDB avec Lambda en utilisant Ruby.

```
require 'json'

def lambda_handler(event:, context:)
  event['events'].each do |record|
    log_document_db_event(record)
  end
  'OK'
end

def log_document_db_event(record)
  event_data = record['event'] || {}
  operation_type = event_data['operationType'] || 'Unknown'
  db = event_data.dig('ns', 'db') || 'Unknown'
  collection = event_data.dig('ns', 'coll') || 'Unknown'
  full_document = event_data['fullDocument'] || {}

  puts "Operation type: #{operation_type}"
  puts "db: #{db}"
  puts "collection: #{collection}"
  puts "Full document: #{JSON.pretty_generate(full_document)}"
end
```

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon DocumentDB avec Lambda en utilisant Rust.

```
use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::documentdb::{DocumentDbEvent, DocumentDbInnerEvent},
   };


// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
//tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<DocumentDbEvent>) ->Result<(),
 Error> {

    tracing::info!("Event Source ARN: {:?}", event.payload.event_source_arn);
    tracing::info!("Event Source: {:?}", event.payload.event_source);

    let records = &event.payload.events;

    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    for record in records{
        log_document_db_event(record);
```

```
    }

    tracing::info!("Document db records processed");

    // Prepare the response
    Ok(())

}

fn log_document_db_event(record: &DocumentDbInnerEvent)-> Result<(), Error>{
    tracing::info!("Change Event: {:?}", record.event);

    Ok(())

}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())

}
```

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Invocation d’une fonction Lambda à partir d’un déclencheur Amazon MSK

Les exemples de code suivants illustrent comment implémenter une fonction Lambda qui reçoit un événement déclenché par la réception d’enregistrements à partir d’un cluster Amazon MSK. La fonction récupère les données utiles MSK et journalise le contenu de l’enregistrement.

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon MSK avec Lambda en utilisant .NET.

```
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KafkaEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace MSKLambda;

public class Function
{


    /// <param name="input">The event for the Lambda function handler to
 process.</param>
    /// <param name="context">The ILambdaContext that provides methods for
 logging and describing the Lambda environment.</param>
    /// <returns></returns>
    public void FunctionHandler(KafkaEvent evnt, ILambdaContext context)
    {

        foreach (var record in evnt.Records)
        {
            Console.WriteLine("Key:" + record.Key);
            foreach (var eventRecord in record.Value)
            {
                var valueBytes = eventRecord.Value.ToArray();
                var valueText = Encoding.UTF8.GetString(valueBytes);
```

```
                Console.WriteLine("Message:" + valueText);
            }
        }
    }


}
```

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon MSK avec Lambda en utilisant Go.

```
package main

import (
 "encoding/base64"
 "fmt"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.KafkaEvent) {
 for key, records := range event.Records {
  fmt.Println("Key:", key)

  for _, record := range records {
   fmt.Println("Record:", record)

   decodedValue, _ := base64.StdEncoding.DecodeString(record.Value)
   message := string(decodedValue)
```

```
   fmt.Println("Message:", message)
  }
 }
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon MSK avec Lambda en utilisant Java.

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KafkaEvent;
import com.amazonaws.services.lambda.runtime.events.KafkaEvent.KafkaEventRecord;

import java.util.Base64;
import java.util.Map;

public class Example implements RequestHandler<KafkaEvent, Void> {

    @Override
    public Void handleRequest(KafkaEvent event, Context context) {
        for (Map.Entry<String, java.util.List<KafkaEventRecord>> entry :
 event.getRecords().entrySet()) {
            String key = entry.getKey();
            System.out.println("Key: " + key);

            for (KafkaEventRecord record : entry.getValue()) {
                System.out.println("Record: " + record);
```

```
                byte[] value = Base64.getDecoder().decode(record.getValue());
                String message = new String(value);
                System.out.println("Message: " + message);
            }
        }

        return null;
    }
}
```

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement Amazon MSK avec JavaScript Lambda à l'aide de.

```
exports.handler = async (event) => {
    // Iterate through keys
    for (let key in event.records) {
      console.log('Key: ', key)
      // Iterate through records
      event.records[key].map((record) => {
        console.log('Record: ', record)
        // Decode base64
        const msg = Buffer.from(record.value, 'base64').toString()
        console.log('Message:', msg)
      })
    }
}
```

Utilisation d'un événement Amazon MSK avec TypeScript Lambda à l'aide de.

```
import { MSKEvent, Context } from "aws-lambda";
```

```
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "msk-handler-sample",
});

export const handler = async (
  event: MSKEvent,
  context: Context
): Promise<void> => {
  for (const [topic, topicRecords] of Object.entries(event.records)) {
    logger.info(`Processing key: ${topic}`);

    // Process each record in the partition
    for (const record of topicRecords) {
      try {
        // Decode the message value from base64
        const decodedMessage = Buffer.from(record.value, 'base64').toString();

        logger.info({
          message: decodedMessage
        });
      }
      catch (error) {
        logger.error('Error processing event', { error });
        throw error;
      }
    };
  }
}
```

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon MSK avec Lambda en utilisant PHP.

```
<?php
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

// using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kafka\KafkaEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): void
    {
        $kafkaEvent = new KafkaEvent($event);
        $this->logger->info("Processing records");
        $records = $kafkaEvent->getRecords();
```

```
        foreach ($records as $record) {
            try {
                $key = $record->getKey();
                $this->logger->info("Key: $key");

                $values = $record->getValue();
                $this->logger->info(json_encode($values));

                foreach ($values as $value) {
                    $this->logger->info("Value: $value");
                }

            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon MSK avec Lambda en utilisant Python.

```
import base64

def lambda_handler(event, context):
    # Iterate through keys
```

```
    for key in event['records']:
        print('Key:', key)
        # Iterate through records
        for record in event['records'][key]:
            print('Record:', record)
            # Decode base64
            msg = base64.b64decode(record['value']).decode('utf-8')
            print('Message:', msg)
```

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon MSK avec Lambda en utilisant Ruby.

```
require 'base64'

def lambda_handler(event:, context:)
  # Iterate through keys
  event['records'].each do |key, records|
    puts "Key: #{key}"

    # Iterate through records
    records.each do |record|
      puts "Record: #{record}"

      # Decode base64
      msg = Base64.decode64(record['value'])
      puts "Message: #{msg}"
    end
  end
end
```

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement Amazon MSK avec Lambda en utilisant Rust.

```
use aws_lambda_events::event::kafka::KafkaEvent;
use lambda_runtime::{run, service_fn, tracing, Error, LambdaEvent};
use base64::prelude::*;
use serde_json::{Value};
use tracing::{info};

/// Pre-Requisites:
/// 1. Install Cargo Lambda - see https://www.cargo-lambda.info/guide/getting-
started.html
/// 2. Add packages tracing, tracing-subscriber, serde_json, base64
///
/// This is the main body for the function.
/// Write your code inside it.
/// There are some code example in the following URLs:
/// - https://github.com/awslabs/aws-lambda-rust-runtime/tree/main/examples
/// - https://github.com/aws-samples/serverless-rust-demo/

async fn function_handler(event: LambdaEvent<KafkaEvent>) -> Result<Value, Error>
 {

    let payload = event.payload.records;

    for (_name, records) in payload.iter() {

        for record in records {

         let record_text = record.value.as_ref().ok_or("Value is None")?;
         info!("Record: {}", &record_text);

         // perform Base64 decoding
         let record_bytes = BASE64_STANDARD.decode(record_text)?;
```

```
            let message = std::str::from_utf8(&record_bytes)?;

            info!("Message: {}", message);
          }

    }

    Ok(().into())
}

#[tokio::main]
async fn main() -> Result<(), Error> {

    // required to enable CloudWatch error logging by the runtime
    tracing::init_default_subscriber();
    info!("Setup CW subscriber!");

    run(service_fn(function_handler)).await
}
```

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Invoquer une fonction Lambda à partir d'un déclencheur Amazon S3

Les exemples de code suivants montrent comment implémenter une fonction Lambda qui reçoit un événement déclenché par le chargement d'un objet vers un compartiment S3. La fonction extrait le nom du compartiment S3 et la clé de l'objet à partir du paramètre d'événement et appelle l'API Amazon S3 pour récupérer et consigner le type de contenu de l'objet.

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement S3 avec Lambda en utilisant .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.S3;
using System;
using Amazon.Lambda.S3Events;
using System.Web;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace S3Integration
{
    public class Function
    {
        private static AmazonS3Client _s3Client;
        public Function() : this(null)
        {
        }

        internal Function(AmazonS3Client s3Client)
        {
            _s3Client = s3Client ?? new AmazonS3Client();
        }

        public async Task<string> Handler(S3Event evt, ILambdaContext context)
        {
            try
            {
                if (evt.Records.Count <= 0)
                {
                    context.Logger.LogLine("Empty S3 Event received");
                    return string.Empty;
                }

                var bucket = evt.Records[0].S3.Bucket.Name;
                var key = HttpUtility.UrlDecode(evt.Records[0].S3.Object.Key);

                context.Logger.LogLine($"Request is for {bucket} and {key}");
```

```
                var objectResult = await _s3Client.GetObjectAsync(bucket, key);

                context.Logger.LogLine($"Returning {objectResult.Key}");

                return objectResult.Key;
            }
            catch (Exception e)
            {
                context.Logger.LogLine($"Error processing request -
 {e.Message}");

                return string.Empty;
            }
        }
    }
}
```

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement S3 avec Lambda en utilisant Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/s3"
```

```
)

func handler(ctx context.Context, s3Event events.S3Event) error {
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Printf("failed to load default config: %s", err)
  return err
 }
 s3Client := s3.NewFromConfig(sdkConfig)

 for _, record := range s3Event.Records {
  bucket := record.S3.Bucket.Name
  key := record.S3.Object.URLDecodedKey
  headOutput, err := s3Client.HeadObject(ctx, &s3.HeadObjectInput{
   Bucket: &bucket,
   Key:    &key,
  })
  if err != nil {
   log.Printf("error getting head of object %s/%s: %s", bucket, key, err)
   return err
  }
  log.Printf("successfully retrieved %s/%s of type %s", bucket, key,
 *headOutput.ContentType)
 }

 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement S3 avec Lambda en utilisant Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import software.amazon.awssdk.services.s3.model.HeadObjectRequest;
import software.amazon.awssdk.services.s3.model.HeadObjectResponse;
import software.amazon.awssdk.services.s3.S3Client;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;
import
 com.amazonaws.services.lambda.runtime.events.models.s3.S3EventNotification.S3EventNotifi

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Handler implements RequestHandler<S3Event, String> {
    private static final Logger logger = LoggerFactory.getLogger(Handler.class);
    @Override
    public String handleRequest(S3Event s3event, Context context) {
        try {
          S3EventNotificationRecord record = s3event.getRecords().get(0);
          String srcBucket = record.getS3().getBucket().getName();
          String srcKey = record.getS3().getObject().getUrlDecodedKey();

          S3Client s3Client = S3Client.builder().build();
          HeadObjectResponse headObject = getHeadObject(s3Client, srcBucket,
 srcKey);

          logger.info("Successfully retrieved " + srcBucket + "/" + srcKey + " of
 type " + headObject.contentType());

          return "Ok";
        } catch (Exception e) {
          throw new RuntimeException(e);
        }
    }

    private HeadObjectResponse getHeadObject(S3Client s3Client, String bucket,
 String key) {
        HeadObjectRequest headObjectRequest = HeadObjectRequest.builder()
```

```
                .bucket(bucket)
                .key(key)
                .build();
        return s3Client.headObject(headObjectRequest);
    }
}
```

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement S3 avec Lambda en utilisant. JavaScript

```
import { S3Client, HeadObjectCommand } from "@aws-sdk/client-s3";

const client = new S3Client();

export const handler = async (event, context) => {

    // Get the object from the event and show its content type
    const bucket = event.Records[0].s3.bucket.name;
    const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g,
 ' '));

    try {
        const { ContentType } = await client.send(new HeadObjectCommand({
            Bucket: bucket,
            Key: key,
        }));

        console.log('CONTENT TYPE:', ContentType);
        return ContentType;

    } catch (err) {
        console.log(err);
```

```
        const message = `Error getting object ${key} from bucket ${bucket}. Make
 sure they exist and your bucket is in the same region as this function.`;
        console.log(message);
        throw new Error(message);
    }
};
```

Consommation d'un événement S3 avec Lambda en utilisant. TypeScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { S3Event } from 'aws-lambda';
import { S3Client, HeadObjectCommand } from '@aws-sdk/client-s3';

const s3 = new S3Client({ region: process.env.AWS_REGION });

export const handler = async (event: S3Event): Promise<string | undefined> => {
  // Get the object from the event and show its content type
  const bucket = event.Records[0].s3.bucket.name;
  const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, '
 '));
  const params = {
    Bucket: bucket,
    Key: key,
  };
  try {
    const { ContentType } = await s3.send(new HeadObjectCommand(params));
    console.log('CONTENT TYPE:', ContentType);
    return ContentType;
  } catch (err) {
    console.log(err);
    const message = `Error getting object ${key} from bucket ${bucket}. Make sure
 they exist and your bucket is in the same region as this function.`;
    console.log(message);
    throw new Error(message);
  }
};
```

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement S3 avec Lambda à l'aide de PHP.

```
<?php

use Bref\Context\Context;
use Bref\Event\S3\S3Event;
use Bref\Event\S3\S3Handler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';


class Handler extends S3Handler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    public function handleS3(S3Event $event, Context $context) : void
    {
        $this->logger->info("Processing S3 records");

        // Get the object from the event and show its content type
        $records = $event->getRecords();

        foreach ($records as $record)
        {
            $bucket = $record->getBucket()->getName();
            $key = urldecode($record->getObject()->getKey());

            try {
```

```
                $fileSize = urldecode($record->getObject()->getSize());
                echo "File Size: " . $fileSize . "\n";
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                echo $e->getMessage() . "\n";
                echo 'Error getting object ' . $key . ' from bucket ' .
 $bucket . '. Make sure they exist and your bucket is in the same region as this
 function.' . "\n";
                throw $e;
            }
        }
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement S3 avec Lambda en utilisant Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import json
import urllib.parse
import boto3

print('Loading function')

s3 = boto3.client('s3')


def lambda_handler(event, context):
```

```
    #print("Received event: " + json.dumps(event, indent=2))

    # Get the object from the event and show its content type
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']['key'],
encoding='utf-8')
    try:
        response = s3.get_object(Bucket=bucket, Key=key)
        print("CONTENT TYPE: " + response['ContentType'])
        return response['ContentType']
    except Exception as e:
        print(e)
        print('Error getting object {} from bucket {}. Make sure they exist and
your bucket is in the same region as this function.'.format(key, bucket))
        raise e
```

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement S3 avec Lambda à l'aide de Ruby.

```
require 'json'
require 'uri'
require 'aws-sdk'

puts 'Loading function'

def lambda_handler(event:, context:)
  s3 = Aws::S3::Client.new(region: 'region') # Your AWS region
  # puts "Received event: #{JSON.dump(event)}"

  # Get the object from the event and show its content type
  bucket = event['Records'][0]['s3']['bucket']['name']
```

```
  key = URI.decode_www_form_component(event['Records'][0]['s3']['object']['key'],
 Encoding::UTF_8)
  begin
    response = s3.get_object(bucket: bucket, key: key)
    puts "CONTENT TYPE: #{response.content_type}"
    return response.content_type
  rescue StandardError => e
    puts e.message
    puts "Error getting object #{key} from bucket #{bucket}. Make sure they exist
 and your bucket is in the same region as this function."
    raise e
  end
end
```

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement S3 avec Lambda en utilisant Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::s3::S3Event;
use aws_sdk_s3::{Client};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};


/// Main function
#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();
```

```
    // Initialize the AWS SDK for Rust
    let config = aws_config::load_from_env().await;
    let s3_client = Client::new(&config);

    let res = run(service_fn(|request: LambdaEvent<S3Event>| {
        function_handler(&s3_client, request)
    })).await;

    res
}

async fn function_handler(
    s3_client: &Client,
    evt: LambdaEvent<S3Event>
) -> Result<(), Error> {
    tracing::info!(records = ?evt.payload.records.len(), "Received request from
 SQS");

    if evt.payload.records.len() == 0 {
        tracing::info!("Empty S3 event received");
    }

    let bucket = evt.payload.records[0].s3.bucket.name.as_ref().expect("Bucket
 name to exist");
    let key = evt.payload.records[0].s3.object.key.as_ref().expect("Object key to
 exist");

    tracing::info!("Request is for {} and object {}", bucket, key);

    let s3_get_object_result = s3_client
        .get_object()
        .bucket(bucket)
        .key(key)
        .send()
        .await;

    match s3_get_object_result {
        Ok(_) => tracing::info!("S3 Get Object success, the s3GetObjectResult
 contains a 'body' property of type ByteStream"),
        Err(_) => tracing::info!("Failure with S3 Get Object request")
    }

    Ok(())
```

```
}
```

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Invocation d'une fonction lambda à partir d'un déclencheur Amazon SNS

Les exemples de code suivants montrent comment implémenter une fonction Lambda qui reçoit un événement déclenché par la réception de messages à partir d'une rubrique SNS. La fonction extrait les messages du paramètre d'événement et consigne le contenu de chaque message.

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement SNS avec Lambda à l'aide de .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SNSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SnsIntegration;

public class Function
{
    public async Task FunctionHandler(SNSEvent evnt, ILambdaContext context)
    {
```

```
        foreach (var record in evnt.Records)
        {
            await ProcessRecordAsync(record, context);
        }
        context.Logger.LogInformation("done");
    }

    private async Task ProcessRecordAsync(SNSEvent.SNSRecord record,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed record
 {record.Sns.Message}");

            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }
    }
}
```

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement SNS avec Lambda à l'aide de Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
```

```
package main

import (
 "context"
 "fmt"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, snsEvent events.SNSEvent) {
 for _, record := range snsEvent.Records {
  processMessage(record)
 }
 fmt.Println("done")
}

func processMessage(record events.SNSEventRecord) {
 message := record.SNS.Message
 fmt.Printf("Processed message: %s\n", message)
 // TODO: Process your record here
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement SNS avec Lambda à l'aide de Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
```

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SNSEvent;
import com.amazonaws.services.lambda.runtime.events.SNSEvent.SNSRecord;


import java.util.Iterator;
import java.util.List;

public class SNSEventHandler implements RequestHandler<SNSEvent, Boolean> {
    LambdaLogger logger;

    @Override
    public Boolean handleRequest(SNSEvent event, Context context) {
        logger = context.getLogger();
        List<SNSRecord> records = event.getRecords();
        if (!records.isEmpty()) {
            Iterator<SNSRecord> recordsIter = records.iterator();
            while (recordsIter.hasNext()) {
                processRecord(recordsIter.next());
            }
        }
        return Boolean.TRUE;
    }

    public void processRecord(SNSRecord record) {
        try {
            String message = record.getSNS().getMessage();
            logger.log("message: " + message);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

}
```

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement SNS avec JavaScript Lambda en utilisant.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record) {
  try {
    const message = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

Consommation d'un événement SNS avec TypeScript Lambda en utilisant.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SNSEvent, Context, SNSHandler, SNSEventRecord } from "aws-lambda";

export const functionHandler: SNSHandler = async (
  event: SNSEvent,
  context: Context
```

```
): Promise<void> => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record: SNSEventRecord): Promise<any> {
  try {
    const message: string = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement SNS avec Lambda à l'aide de PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

/*
Since native PHP support for AWS Lambda is not available, we are utilizing Bref's
 PHP functions runtime for AWS Lambda.
For more information on Bref's PHP runtime for Lambda, refer to: https://bref.sh/
docs/runtimes/function

Another approach would be to create a custom runtime.
```

```
A practical example can be found here: https://aws.amazon.com/blogs/apn/aws-
lambda-custom-runtime-for-php-a-practical-example/
*/

// Additional composer packages may be required when using Bref or any other PHP
 functions runtime.
// require __DIR__ . '/vendor/autoload.php';

use Bref\Context\Context;
use Bref\Event\Sns\SnsEvent;
use Bref\Event\Sns\SnsHandler;

class Handler extends SnsHandler
{
    public function handleSns(SnsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $message = $record->getMessage();

            // TODO: Implement your custom processing logic here
            // Any exception thrown will be logged and the invocation will be
 marked as failed

            echo "Processed Message: $message" . PHP_EOL;
        }
    }
}

return new Handler();
```

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement SNS avec Lambda à l'aide de Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for record in event['Records']:
        process_message(record)
    print("done")

def process_message(record):
    try:
        message = record['Sns']['Message']
        print(f"Processed message {message}")
        # TODO; Process your record here

    except Exception as e:
        print("An error occurred")
        raise e
```

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement SNS avec Lambda à l'aide de Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].map { |record| process_message(record) }
end

def process_message(record)
  message = record['Sns']['Message']
  puts("Processing message: #{message}")
rescue StandardError => e
  puts("Error processing message: #{e}")
```

```
  raise
end
```

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement SNS avec Lambda à l'aide de Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sns::SnsEvent;
use aws_lambda_events::sns::SnsRecord;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
use tracing::info;

// Built with the following dependencies:
//  aws_lambda_events = { version = "0.10.0", default-features = false, features
 = ["sns"] }
//  lambda_runtime = "0.8.1"
//  tokio = { version = "1", features = ["macros"] }
//  tracing = { version = "0.1", features = ["log"] }
//  tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }

async fn function_handler(event: LambdaEvent<SnsEvent>) -> Result<(), Error> {
    for event in event.payload.records {
        process_record(&event)?;
    }

    Ok(())
}

fn process_record(record: &SnsRecord) -> Result<(), Error> {
    info!("Processing SNS Message: {}", record.sns.message);
```

```
    // Implement your record handling code here.

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Invoquer une fonction Lambda à partir d'un déclencheur Amazon SQS

Les exemples de code suivants montrent comment implémenter une fonction Lambda qui reçoit un événement déclenché par la réception de messages à partir d'une file d'attente SQS. La fonction extrait les messages du paramètre d'événement et consigne le contenu de chaque message.

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement SQS avec Lambda en utilisant .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
```

```
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SqsIntegrationSampleCode
{
    public async Task FunctionHandler(SQSEvent evnt, ILambdaContext context)
    {
        foreach (var message in evnt.Records)
        {
            await ProcessMessageAsync(message, context);
        }

        context.Logger.LogInformation("done");
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed message {message.Body}");

            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }

    }
}
```

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement SQS avec Lambda à l'aide de Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package integration_sqs_to_lambda

import (
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.SQSEvent) error {
 for _, record := range event.Records {
  err := processMessage(record)
  if err != nil {
   return err
  }
 }
 fmt.Println("done")
 return nil
}

func processMessage(record events.SQSMessage) error {
 fmt.Printf("Processed message %s\n", record.Body)
 // TODO: Do interesting work based on the new message
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement SQS avec Lambda à l'aide de Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSEvent.SQSMessage;

public class Function implements RequestHandler<SQSEvent, Void> {
    @Override
    public Void handleRequest(SQSEvent sqsEvent, Context context) {
        for (SQSMessage msg : sqsEvent.getRecords()) {
            processMessage(msg, context);
        }
        context.getLogger().log("done");
        return null;
    }

    private void processMessage(SQSMessage msg, Context context) {
        try {
            context.getLogger().log("Processed message " + msg.getBody());

            // TODO: Do interesting work based on the new message

        } catch (Exception e) {
            context.getLogger().log("An error occurred");
            throw e;
        }
```

```
    }
}
```

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement SQS avec JavaScript Lambda en utilisant.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message) {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

Consommation d'un événement SQS avec TypeScript Lambda en utilisant.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, Context, SQSHandler, SQSRecord } from "aws-lambda";
```

```
export const functionHandler: SQSHandler = async (
  event: SQSEvent,
  context: Context
): Promise<void> => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message: SQSRecord): Promise<any> {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement SQS avec Lambda à l'aide de PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
```

```
use Bref\Event\InvalidLambdaEvent;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $body = $record->getBody();
            // TODO: Do interesting work based on the new message
        }
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement SQS avec Lambda à l'aide de Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for message in event['Records']:
        process_message(message)
    print("done")

def process_message(message):
    try:
        print(f"Processed message {message['body']}")
        # TODO: Do interesting work based on the new message
    except Exception as err:
        print("An error occurred")
        raise err
```

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Utilisation d'un événement SQS avec Lambda à l'aide de Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].each do |message|
    process_message(message)
  end
  puts "done"
end

def process_message(message)
  begin
    puts "Processed message #{message['body']}"
    # TODO: Do interesting work based on the new message
```

```
  rescue StandardError => err
    puts "An error occurred"
    raise err
  end
end
```

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Consommation d'un événement SQS avec Lambda à l'aide de Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sqs::SqsEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<SqsEvent>) -> Result<(), Error> {
    event.payload.records.iter().for_each(|record| {
        // process the record
        tracing::info!("Message body: {}",
 record.body.as_deref().unwrap_or_default())
    });

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
```

```
        .init();

    run(service_fn(function_handler)).await
}
```

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Signalement des échecs d'articles par lots pour les fonctions Lambda à l'aide d'un déclencheur Kinesis

Les exemples de code suivants montrent comment implémenter une réponse par lots partielle pour les fonctions Lambda qui reçoivent des événements d'un flux Kinesis. La fonction signale les défaillances échecs d'articles par lots dans la réponse, en indiquant à Lambda de réessayer ces messages ultérieurement.

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots Kinesis avec Lambda à l'aide de .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using System.Text.Json.Serialization;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
```

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegration;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task<StreamsEventResponse> FunctionHandler(KinesisEvent evnt,
 ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return new StreamsEventResponse();
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                return new StreamsEventResponse
                {
                    BatchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>
                    {
                        new StreamsEventResponse.BatchItemFailure
 { ItemIdentifier = record.Kinesis.SequenceNumber }
                    }
```

```
                };
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
        return new StreamsEventResponse();
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}

public class StreamsEventResponse
{
    [JsonPropertyName("batchItemFailures")]
    public IList<BatchItemFailure> BatchItemFailures { get; set; }
    public class BatchItemFailure
    {
        [JsonPropertyName("itemIdentifier")]
        public string ItemIdentifier { get; set; }
    }
}
```

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots Kinesis avec Lambda à l'aide de Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
```

```
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, record := range kinesisEvent.Records {
  curRecordSequenceNumber := ""

  // Process your record
  if /* Your record processing condition here */ {
   curRecordSequenceNumber = record.Kinesis.SequenceNumber
  }

  // Add a condition to check if the record processing failed
  if curRecordSequenceNumber != "" {
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": curRecordSequenceNumber})
  }
 }

 kinesisBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
 }
 return kinesisBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots Kinesis avec Lambda à l'aide de Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessKinesisRecords implements RequestHandler<KinesisEvent,
 StreamsEventResponse> {

    @Override
    public StreamsEventResponse handleRequest(KinesisEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (KinesisEvent.KinesisEventRecord kinesisEventRecord :
 input.getRecords()) {
            try {
                //Process your record
                KinesisEvent.Record kinesisRecord =
 kinesisEventRecord.getKinesis();
                curRecordSequenceNumber = kinesisRecord.getSequenceNumber();

            } catch (Exception e) {
```

```
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse(batchItemFailures);
    }
}
```

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots Kinesis avec Lambda à l'aide de JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
```

```
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

Signaler les défaillances d'éléments de lot Kinesis avec Lambda à l'aide de. TypeScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
  KinesisStreamBatchResponse,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<KinesisStreamBatchResponse> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
```

```
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  logger.info(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots Kinesis avec Lambda à l'aide de PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php
```

```
# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $kinesisEvent = new KinesisEvent($event);
        $this->logger->info("Processing records");
        $records = $kinesisEvent->getRecords();

        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
```

```
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots Kinesis avec Lambda à l'aide de Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["kinesis"]["sequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots Kinesis avec Lambda à l'aide de Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  batch_item_failures = []

  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue StandardError => err
      puts "An error occurred #{err}"
      # Since we are working with streams, we can return the failed item
 immediately.
      # Lambda will immediately begin to retry processing from this failed item
 onwards.
      return { batchItemFailures: [{ itemIdentifier: record['kinesis']
['sequenceNumber'] }] }
    end
  end

  puts "Successfully processed #{event['Records'].length} records."
  { batchItemFailures: batch_item_failures }
end
```

```
def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('utf-8')
  # Placeholder for actual async work
  sleep(1)
  data
end
```

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots Kinesis avec Lambda à l'aide de Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::kinesis::KinesisEvent,
    kinesis::KinesisEventRecord,
    streams::{KinesisBatchItemFailure, KinesisEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) ->
 Result<KinesisEventResponse, Error> {
    let mut response = KinesisEventResponse {
        batch_item_failures: vec![],
    };

    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in &event.payload.records {
        tracing::info!(
            "EventId: {}",
```

```
            record.event_id.as_deref().unwrap_or_default()
        );

        let record_processing_result = process_record(record);

        if record_processing_result.is_err() {
            response.batch_item_failures.push(KinesisBatchItemFailure {
                item_identifier: record.kinesis.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(response)
}

fn process_record(record: &KinesisEventRecord) -> Result<(), Error> {
    let record_data = std::str::from_utf8(record.kinesis.data.as_slice());

    if let Some(err) = record_data.err() {
        tracing::error!("Error: {}", err);
        return Err(Error::from(err));
    }

    let record_data = record_data.unwrap_or_default();

    // do something interesting with the data
    tracing::info!("Data: {}", record_data);

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
```

```
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Signalement des échecs d'articles par lots pour les fonctions Lambda à l'aide d'un déclencheur DynamoDB

Les exemples de code suivants illustrent comment implémenter une réponse partielle par lots pour les fonctions Lambda qui reçoivent des événements à partir d'un flux DynamoDB. La fonction signale les défaillances échecs d'articles par lots dans la réponse, en indiquant à Lambda de réessayer ces messages ultérieurement.

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots DynamoDB avec Lambda à l'aide de .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
```

```
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public StreamsEventResponse FunctionHandler(DynamoDBEvent dynamoEvent,
 ILambdaContext context)

    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");
        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>();
        StreamsEventResponse streamsEventResponse = new StreamsEventResponse();

        foreach (var record in dynamoEvent.Records)
        {
            try
            {
                var sequenceNumber = record.Dynamodb.SequenceNumber;
                context.Logger.LogInformation(sequenceNumber);
            }
            catch (Exception ex)
            {
                context.Logger.LogError(ex.Message);
                batchItemFailures.Add(new StreamsEventResponse.BatchItemFailure()
 { ItemIdentifier = record.Dynamodb.SequenceNumber });
            }
        }

        if (batchItemFailures.Count > 0)
        {
            streamsEventResponse.BatchItemFailures = batchItemFailures;
        }

        context.Logger.LogInformation("Stream processing complete.");
        return streamsEventResponse;
    }
```

```
}
```

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots DynamoDB avec Lambda à l'aide de Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

type BatchItemFailure struct {
 ItemIdentifier string `json:"ItemIdentifier"`
}

type BatchResult struct {
 BatchItemFailures []BatchItemFailure `json:"BatchItemFailures"`
}

func HandleRequest(ctx context.Context, event events.DynamoDBEvent)
 (*BatchResult, error) {
 var batchItemFailures []BatchItemFailure
 curRecordSequenceNumber := ""

 for _, record := range event.Records {
  // Process your record
  curRecordSequenceNumber = record.Change.SequenceNumber
 }
```

```
 if curRecordSequenceNumber != "" {
  batchItemFailures = append(batchItemFailures, BatchItemFailure{ItemIdentifier:
 curRecordSequenceNumber})
 }

 batchResult := BatchResult{
  BatchItemFailures: batchItemFailures,
 }

 return &batchResult, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots DynamoDB avec Lambda à l'aide de Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import com.amazonaws.services.lambda.runtime.events.models.dynamodb.StreamRecord;

import java.util.ArrayList;
import java.util.List;

public class ProcessDynamodbRecords implements RequestHandler<DynamodbEvent,
 StreamsEventResponse> {
```

```
    @Override
    public StreamsEventResponse handleRequest(DynamodbEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (DynamodbEvent.DynamodbStreamRecord dynamodbStreamRecord :
 input.getRecords()) {
          try {
                //Process your record
                StreamRecord dynamodbRecord = dynamodbStreamRecord.getDynamodb();
                curRecordSequenceNumber = dynamodbRecord.getSequenceNumber();

            } catch (Exception e) {
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse();
    }
}
```

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signaler les défaillances d'éléments de lot DynamoDB avec Lambda à l'aide de. JavaScript

```
export const handler = async (event) => {
  const records = event.Records;
  let curRecordSequenceNumber = "";

  for (const record of records) {
    try {
      // Process your record
      curRecordSequenceNumber = record.dynamodb.SequenceNumber;
    } catch (e) {
      // Return failed record's sequence number
      return { batchItemFailures: [{ itemIdentifier:
 curRecordSequenceNumber }] };
    }
  }

  return { batchItemFailures: [] };
};
```

Signaler les défaillances d'éléments de lot DynamoDB avec Lambda à l'aide de. TypeScript

```
import {
  DynamoDBBatchResponse,
  DynamoDBBatchItemFailure,
  DynamoDBStreamEvent,
} from "aws-lambda";

export const handler = async (
  event: DynamoDBStreamEvent
): Promise<DynamoDBBatchResponse> => {
  const batchItemFailures: DynamoDBBatchItemFailure[] = [];
  let curRecordSequenceNumber;

  for (const record of event.Records) {
    curRecordSequenceNumber = record.dynamodb?.SequenceNumber;

    if (curRecordSequenceNumber) {
      batchItemFailures.push({
        itemIdentifier: curRecordSequenceNumber,
      });
    }
```

```
  }

  return { batchItemFailures: batchItemFailures };
};
```

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots DynamoDB avec Lambda à l'aide de PHP.

```
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
```

```
    {
        $dynamoDbEvent = new DynamoDbEvent($event);
        $this->logger->info("Processing records");

        $records = $dynamoDbEvent->getRecords();
        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots DynamoDB avec Lambda à l'aide de Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["dynamodb"]["SequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots DynamoDB avec Lambda à l'aide de Ruby.

```
def lambda_handler(event:, context:)
    records = event["Records"]
    cur_record_sequence_number = ""

    records.each do |record|
      begin
        # Process your record
        cur_record_sequence_number = record["dynamodb"]["SequenceNumber"]
      rescue StandardError => e
        # Return failed record's sequence number
        return {"batchItemFailures" => [{"itemIdentifier" =>
 cur_record_sequence_number}]}
      end
    end

    {"batchItemFailures" => []}
  end
```

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots DynamoDB avec Lambda à l'aide de Rust.

```
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord, StreamRecord},
    streams::{DynamoDbBatchItemFailure, DynamoDbEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

/// Process the stream record
fn process_record(record: &EventRecord) -> Result<(), Error> {
    let stream_record: &StreamRecord = &record.change;
```

```
    // process your stream record here...
    tracing::info!("Data: {:?}", stream_record);

    Ok(())
}

/// Main Lambda handler here...
async fn function_handler(event: LambdaEvent<Event>) ->
 Result<DynamoDbEventResponse, Error> {
    let mut response = DynamoDbEventResponse {
        batch_item_failures: vec![],
    };

    let records = &event.payload.records;

    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in records {
        tracing::info!("EventId: {}", record.event_id);

        // Couldn't find a sequence number
        if record.change.sequence_number.is_none() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: Some("".to_string()),
            });
            return Ok(response);
        }

        // Process your record here...
        if process_record(record).is_err() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: record.change.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }
```

```
    tracing::info!("Successfully processed {} record(s)", records.len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

Signalement des échecs d'articles par lots pour les fonctions Lambda à l'aide d'un déclencheur Amazon SQS

Les exemples de code suivants montrent comment implémenter une réponse par lots partielle pour les fonctions Lambda qui reçoivent des événements d'une file d'attente SQS. La fonction signale les défaillances échecs d'articles par lots dans la réponse, en indiquant à Lambda de réessayer ces messages ultérieurement.

.NET

SDK pour .NET

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots SQS avec Lambda à l'aide de .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri
namespace sqsSample;

public class Function
{
    public async Task<SQSBatchResponse> FunctionHandler(SQSEvent evnt,
 ILambdaContext context)
    {
        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 List<SQSBatchResponse.BatchItemFailure>();
        foreach(var message in evnt.Records)
        {
            try
            {
                //process your message
                await ProcessMessageAsync(message, context);
            }
            catch (System.Exception)
            {
                //Add failed message identifier to the batchItemFailures list
                batchItemFailures.Add(new
 SQSBatchResponse.BatchItemFailure{ItemIdentifier=message.MessageId});
            }
```

```
        }
        return new SQSBatchResponse(batchItemFailures);
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        if (String.IsNullOrEmpty(message.Body))
        {
            throw new Exception("No Body in SQS Message.");
        }
        context.Logger.LogInformation($"Processed message {message.Body}");
        // TODO: Do interesting work based on the new message
        await Task.CompletedTask;
    }
}
```

Go

Kit SDK pour Go V2

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots SQS avec Lambda à l'aide de Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, sqsEvent events.SQSEvent)
 (map[string]interface{}, error) {
```

```
 batchItemFailures := []map[string]interface{}{}

 for _, message := range sqsEvent.Records {
  if len(message.Body) > 0 {
   // Your message processing condition here
   fmt.Printf("Successfully processed message: %s\n", message.Body)
  } else {
   // Message processing failed
   fmt.Printf("Failed to process message %s\n", message.MessageId)
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": message.MessageId})
  }
 }

 sqsBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
 }
 return sqsBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK pour Java 2.x

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots SQS avec Lambda à l'aide de Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
```

```
import com.amazonaws.services.lambda.runtime.events.SQSBatchResponse;

import java.util.ArrayList;
import java.util.List;

public class ProcessSQSMessageBatch implements RequestHandler<SQSEvent,
 SQSBatchResponse> {
    @Override
    public SQSBatchResponse handleRequest(SQSEvent sqsEvent, Context context) {
         List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<SQSBatchResponse.BatchItemFailure>();

         for (SQSEvent.SQSMessage message : sqsEvent.getRecords()) {
             try {
                 //process your message
             } catch (Exception e) {
                 //Add failed message identifier to the batchItemFailures list
                 batchItemFailures.add(new
 SQSBatchResponse.BatchItemFailure(message.getMessageId()));
             }
         }
         return new SQSBatchResponse(batchItemFailures);
     }
}
```

JavaScript

SDK pour JavaScript (v3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signaler les défaillances d'éléments de lot SQS avec JavaScript Lambda à l'aide de.

```
// Node.js 20.x Lambda runtime, AWS SDK for Javascript V3
export const handler = async (event, context) => {
    const batchItemFailures = [];
    for (const record of event.Records) {
        try {
```

```
            await processMessageAsync(record, context);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }
    return { batchItemFailures };
};

async function processMessageAsync(record, context) {
    if (record.body && record.body.includes("error")) {
        throw new Error("There is an error in the SQS Message.");
    }
    console.log(`Processed message: ${record.body}`);
}
```

Signaler les défaillances d'éléments de lot SQS avec TypeScript Lambda à l'aide de.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, SQSBatchResponse, Context, SQSBatchItemFailure, SQSRecord }
 from 'aws-lambda';

export const handler = async (event: SQSEvent, context: Context):
 Promise<SQSBatchResponse> => {
    const batchItemFailures: SQSBatchItemFailure[] = [];

    for (const record of event.Records) {
        try {
            await processMessageAsync(record);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }

    return {batchItemFailures: batchItemFailures};
};

async function processMessageAsync(record: SQSRecord): Promise<void> {
    if (record.body && record.body.includes("error")) {
        throw new Error('There is an error in the SQS Message.');
    }
    console.log(`Processed message ${record.body}`);
}
```

PHP

Kit SDK pour PHP

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots SQS avec Lambda à l'aide de PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

use Bref\Context\Context;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        $this->logger->info("Processing SQS records");
        $records = $event->getRecords();
```

```
        foreach ($records as $record) {
            try {
                // Assuming the SQS message is in JSON format
                $message = json_decode($record->getBody(), true);
                $this->logger->info(json_encode($message));
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $this->markAsFailed($record);
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords SQS records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

Kit SDK for Python (Boto3)

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots SQS avec Lambda à l'aide de Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

def lambda_handler(event, context):
    if event:
        batch_item_failures = []
        sqs_batch_response = {}

        for record in event["Records"]:
```

```
            try:
                print(f"Processed message: {record['body']}")
            except Exception as e:
                batch_item_failures.append({"itemIdentifier":
 record['messageId']})

        sqs_batch_response["batchItemFailures"] = batch_item_failures
        return sqs_batch_response
```

Ruby

Kit SDK pour Ruby

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots SQS avec Lambda à l'aide de Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'json'

def lambda_handler(event:, context:)
  if event
    batch_item_failures = []
    sqs_batch_response = {}

    event["Records"].each do |record|
      begin
        # process message
      rescue StandardError => e
        batch_item_failures << {"itemIdentifier" => record['messageId']}
      end
    end

    sqs_batch_response["batchItemFailures"] = batch_item_failures
    return sqs_batch_response
  end
end
```

Rust

SDK pour Rust

> **Note**
>
> Il y en a plus à ce sujet GitHub. Trouvez l'exemple complet et découvrez comment le configurer et l'exécuter dans le référentiel d'exemples sans serveur.

Signalement des échecs d'articles par lots SQS avec Lambda à l'aide de Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::sqs::{SqsBatchResponse, SqsEvent},
    sqs::{BatchItemFailure, SqsMessage},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn process_record(_: &SqsMessage) -> Result<(), Error> {
    Err(Error::from("Error processing message"))
}

async fn function_handler(event: LambdaEvent<SqsEvent>) ->
 Result<SqsBatchResponse, Error> {
    let mut batch_item_failures = Vec::new();
    for record in event.payload.records {
        match process_record(&record).await {
            Ok(_) => (),
            Err(_) => batch_item_failures.push(BatchItemFailure {
                item_identifier: record.message_id.unwrap(),
            }),
        }
    }

    Ok(SqsBatchResponse {
        batch_item_failures,
    })
}
```

```
#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(function_handler)).await
}
```

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

AWS contributions de la communauté pour Lambda

AWS les contributions communautaires sont des exemples qui ont été créés et sont maintenus par plusieurs équipes AWS. Pour faire un commentaire, utilisez le mécanisme fourni dans les référentiels liés.

Exemples

- Création et test d'une application sans serveur

Création et test d'une application sans serveur

Les exemples de code suivants montrent comment créer et tester une application sans serveur à l'aide d'API Gateway avec Lambda et DynamoDB.

.NET

SDK pour .NET

Illustre comment créer et tester une application sans serveur composée d'une API Gateway avec Lambda et DynamoDB à l'aide du kit SDK .NET.

Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Les services utilisés dans cet exemple

- API Gateway
- DynamoDB

- Lambda

Go

Kit SDK pour Go V2

Illustre comment créer et tester une application sans serveur composée d'une API Gateway avec Lambda et DynamoDB à l'aide du kit SDK Go.

Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Les services utilisés dans cet exemple

- API Gateway
- DynamoDB
- Lambda

Java

SDK pour Java 2.x

Illustre comment créer et tester une application sans serveur composée d'une API Gateway avec Lambda et DynamoDB à l'aide du kit SDK Java.

Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Les services utilisés dans cet exemple

- API Gateway
- DynamoDB
- Lambda

Rust

SDK pour Rust

Illustre comment créer et tester une application sans serveur composée d'une API Gateway avec Lambda et DynamoDB à l'aide du kit SDK Rust.

Pour obtenir le code source complet et les instructions de configuration et d'exécution, consultez l'exemple complet sur GitHub.

Les services utilisés dans cet exemple

- API Gateway
- DynamoDB
- Lambda

Pour obtenir la liste complète des guides de développement du AWS SDK et des exemples de code, consultezUtilisation de Lambda avec un SDK AWS. Cette rubrique comprend également des informations sur le démarrage et sur les versions précédentes du kit SDK.

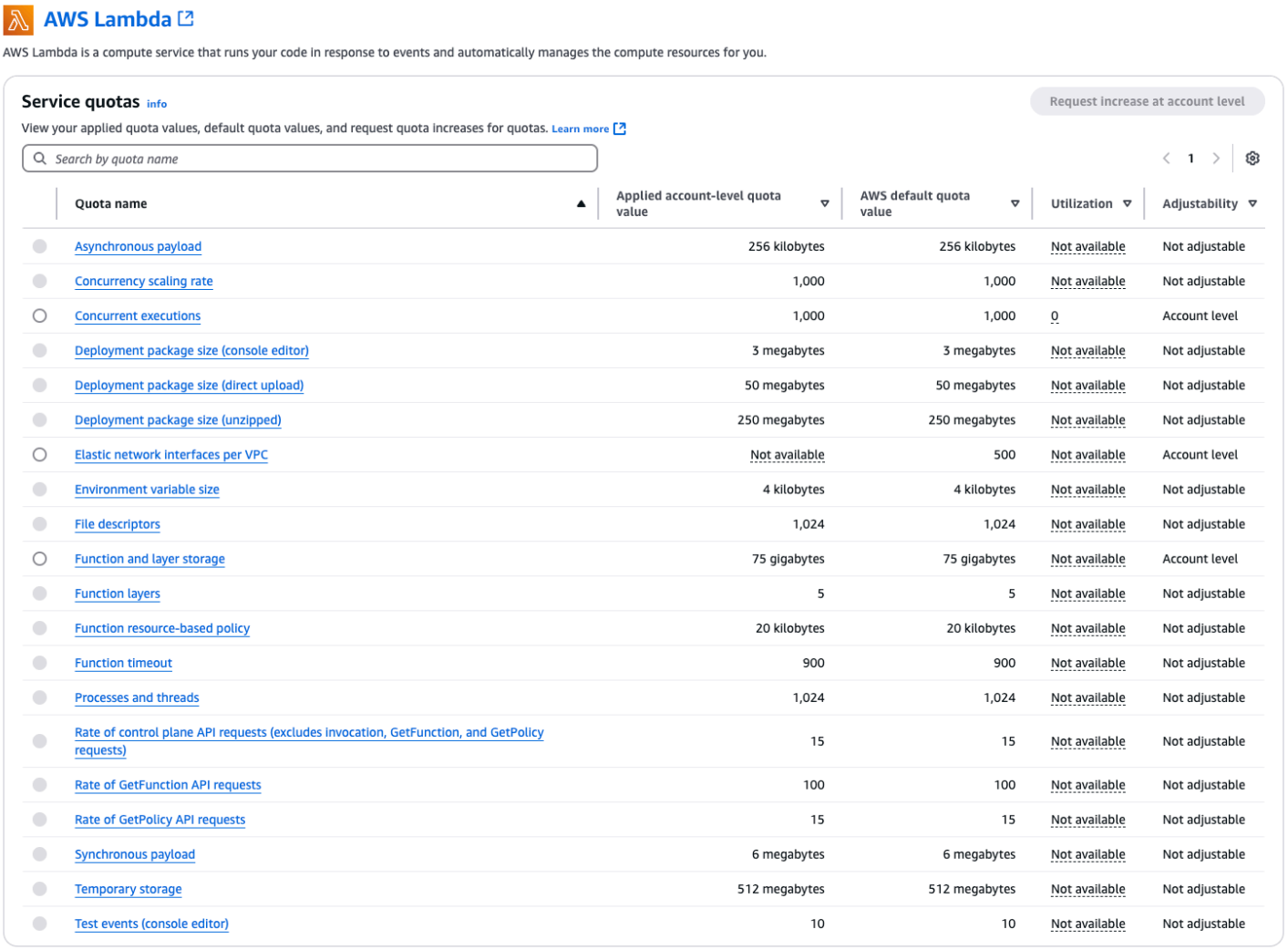
Quotas Lambda

> **Important**
>
> Comptes AWS Les nouveautés ont réduit la simultanéité et les quotas de mémoire. AWS augmente automatiquement ces quotas en fonction de votre utilisation.

AWS Lambda est conçu pour évoluer rapidement afin de répondre à la demande, ce qui permet à vos fonctions de s'adapter au trafic de votre application. Lambda est conçu pour les tâches de calcul de courte durée qui ne conservent pas ou ne dépendent pas de l'état entre les invocations. Le code peut être exécuté pendant 15 minutes au cours d'une seule invocation et chaque fonction peut utiliser jusqu'à 10 240 Mo de mémoire.

Il est important de comprendre les barrières de protection qui sont mises en place pour protéger votre compte et les charges de travail des autres clients. Les quotas de service existent dans tous les AWS services et se composent de limites strictes, que vous ne pouvez pas modifier, et de limites souples, pour lesquelles vous pouvez demander des augmentations. Par défaut, tous les nouveaux comptes se voient attribuer un profil de quota qui permet d'explorer AWS les services.

Pour voir les quotas qui s'appliquent à votre compte, accédez au tableau de bord Service Quotas. Ici, vous pouvez consulter vos quotas de service, demander une augmentation de quota et consulter l'utilisation actuelle. À partir de là, vous pouvez accéder à un AWS service spécifique, tel que Lambda :



AWS Lambda

AWS Lambda is a compute service that runs your code in response to events and automatically manages the compute resources for you.

Service quotas info

Request increase at account level

View your applied quota values, default quota values, and request quota increases for quotas. Learn more

Search by quota name

1

Quota name	Applied account-level quota value	AWS default quota value	Utilization	Adjustability
Asynchronous payload	256 kilobytes	256 kilobytes	Not available	Not adjustable
Concurrency scaling rate	1,000	1,000	Not available	Not adjustable
Concurrent executions	1,000	1,000	0	Account level
Deployment package size (console editor)	3 megabytes	3 megabytes	Not available	Not adjustable
Deployment package size (direct upload)	50 megabytes	50 megabytes	Not available	Not adjustable
Deployment package size (unzipped)	250 megabytes	250 megabytes	Not available	Not adjustable
Elastic network interfaces per VPC	Not available	500	Not available	Account level
Environment variable size	4 kilobytes	4 kilobytes	Not available	Not adjustable
File descriptors	1,024	1,024	Not available	Not adjustable
Function and layer storage	75 gigabytes	75 gigabytes	Not available	Account level
Function layers	5	5	Not available	Not adjustable
Function resource-based policy	20 kilobytes	20 kilobytes	Not available	Not adjustable
Function timeout	900	900	Not available	Not adjustable
Processes and threads	1,024	1,024	Not available	Not adjustable
Rate of control plane API requests (excludes invocation, GetFunction, and GetPolicy requests)	15	15	Not available	Not adjustable
Rate of GetFunction API requests	100	100	Not available	Not adjustable
Rate of GetPolicy API requests	15	15	Not available	Not adjustable
Synchronous payload	6 megabytes	6 megabytes	Not available	Not adjustable
Temporary storage	512 megabytes	512 megabytes	Not available	Not adjustable
Test events (console editor)	10	10	Not available	Not adjustable

Les sections suivantes répertorient les quotas et limites par défaut dans Lambda par catégorie.

Rubriques

- calcul et stockage
- Configuration, déploiement et exécution de fonction
- Requêtes d'API Lambda
- Autres services

calcul et stockage

Lambda définit des quotas pour les ressources de calcul et de stockage que vous pouvez utiliser afin d'exécuter et de stocker des fonctions. Les quotas d'exécutions simultanées et de stockage

s'appliquent par Région AWS. Les quotas d'Interface réseau Elastic (ENI) s'appliquent par cloud privé virtuel (VPC), quelle que soit la Région. Les quotas suivants peuvent être augmentés par rapport à leurs valeurs par défaut. Pour de plus amples informations, veuillez consulter Demande d'augmentation de quota dans le Guide de l'utilisateur Service Quotas.

Ressource	Quota par défaut	Peut être augmentée jusqu'à
Exécutions simultanées	1 000	Dizaines de milliers
Stockage pour les fonctions téléchargées (archives de fichiers .zip) et les couches. Chaque version de fonction et de couche consomme de l'espace de stockage. Pour respecter les bonnes pratiques en matière d'administration du stockage du code, veuillez consulter Surveillance du stockage de code Lambda dans Serverless Land.	75 Go	Téra-octets
Stockage des fonctions définies en tant qu'images de conteneur. Ces images sont stockées dans Amazon ECR.	Consultez Service Quotas Amazon ECR.	
Interfaces réseau Elastic par cloud privé virtuel (VPC) **Note** Ce quota est partagé avec d'autres services, tels qu'Amazon Elastic File System (Amazon EFS). Consultez Quotas Amazon VPC.	500	Milliers

Pour en savoir plus sur la simultanéité et la manière dont Lambda met à l'échelle la simultanéité de votre fonction en réponse au trafic, consultez Présentation de la mise à l'échelle de fonction Lambda.

Configuration, déploiement et exécution de fonction

Les quotas suivants s'appliquent à la configuration, au déploiement et à l'exécution des fonctions. Sauf indication contraire, ils ne peuvent pas être modifiés.

> **Note**
>
> La documentation Lambda, les messages de journal et la console utilisent l'abréviation Mo (plutôt que Mio) pour faire référence à 1024 Ko.

Ressource	Quota
Allocation de mémoire des fonctions	128 Mo à 10 240 Mo, par incréments de 1 Mo Remarque : Lambda alloue de la puissance d'UC en fonction de la quantité de mémoire configurée. Vous pouvez augmenter ou réduire la mémoire et la puissance d'UC allouées à votre fonction à l'aide du paramètre Mémoire (Mo). À 1 769 Mo, une fonction possède l'équival ent d'un vCPU.
Délai d'expiration des fonctions	900 secondes (15 minutes)
Variables d'environnement des fonctions	4 Ko, pour toutes les variables d'environnement associées à la fonction, au total
stratégie de fonction basée sur les ressources	20 Ko
Couches de fonctions	5 couches

Ressource	Quota
Limite d'échelle de simultanéité des fonctions	Pour chaque fonction, 1 000 environnements d'exécution toutes les 10 secondes
Charge utile d'invocation (demande et réponse)	6 Mo chacun pour la demande et la réponse (synchrone) 200 Mo pour chaque réponse diffusée (synchrone) 1 Mo (asynchrone) 1 Mo pour la taille totale combinée des valeurs de ligne de requête et d'en-tête
Bande passante pour les réponses diffusées	Non plafonné pour les 6 premiers Mo de la réponse de votre fonction Pour les réponses supérieures à 6 Mo, 2 MBps pour le reste de la réponse
Taille du package de déploiement (archive de fichiers .zip)	50 Mo (compressés, lors du téléchargement via l'API SDKs Lambda ou). Chargez vos fichiers sur Amazon S3. 50 Mo (en cas de chargement via la console Lambda) 250 Mo La taille maximale du contenu d'un package de déploiement, y compris les couches et les environnements d'exécution personnalisés. (décompressé)

Ressource	Quota
Taille des paramètres de l'image de conteneur	16 Ko
Taille du package du code de l'image de conteneur	10 Go (taille maximale de l'image non compressée, comprenant toutes les couches)
Événements de test (éditeur de console)	10
Stockage dans le répertoire `/tmp`	Entre 512 Mo et 10 240 Mo par incréments de 1 Mo
Descripteurs de fichier	1,024 **Note** Les instances gérées Lambda utilisent les limites de descripteur de fichier par défaut de Bottlerocket. Pour de plus amples informations, veuillez consulter Comprendre l'environnement d'exécution des instances gérées Lambda.

Ressource	Quota
Processus/threads d'exécution	1,024 **Note** Les instances gérées Lambda utilisent les limites de processus et de thread par défaut de Bottleroc ket. Pour de plus amples informations, veuillez consulter Comprendre l'environnement d'exécuti on des instances gérées Lambda.

Requêtes d'API Lambda

Les quotas suivants sont associés aux demandes d'API Lambda.

Ressource	Quota
Demandes d'invocation par fonction par région (synchron e)	Chaque instance de votre environne ment d'exécution peut servir jusqu'à 10 demandes par seconde. En d'autres termes, la limite d'invocat ion totale correspond à 10 fois votre limite de simultanéité. Consultez Présentation de la mise à l'échelle de fonction Lambda.
Demandes d'invocation par fonction par région (asynchro ne)	Chaque instance de votre environne ment d'exécution peut servir un nombre illimité de demandes. En d'autres termes, la limite d'invocat

Ressource	Quota
	ion totale est basée uniquement sur la simultanéité disponible pour votre fonction. Consultez Présentation de la mise à l'échelle de fonction Lambda.
Demandes d'invocation par version ou alias de fonction (demandes par seconde)	10 x simultanéité provisionnée **Note** Ce quota s'applique uniquement aux fonctions qui utilisent la simultanéité provisionnée.
Demandes d'API GetFunction	100 requêtes par seconde. Il ne peut pas être augmenté.
Demandes d'API GetPolicy	15 requêtes par seconde. Il ne peut pas être augmenté.
Reste des demandes d'API du plan de contrôle (à l'exclusion de l'invocation et GetFunction des GetPolicy demandes)	15 requêtes par seconde en tout APIs (et non 15 demandes par seconde par API). Il ne peut pas être augmenté.

Autres services

Les quotas pour d'autres services, tels que Gestion des identités et des accès AWS (IAM), Amazon CloudFront (Lambda @Edge) et Amazon Virtual Private Cloud (Amazon VPC), peuvent avoir un impact sur vos fonctions Lambda. Pour plus d'informations, consultez Quotas Service AWS dans le Référence générale d'Amazon Web Services et Invoquer Lambda avec des événements provenant d'autres services AWS.

De nombreuses applications impliquant Lambda utilisent plusieurs AWS services. Étant donné que les différents services ont des quotas différents pour différentes fonctionnalités, il peut être difficile de gérer ces quotas dans l'ensemble de votre application. API Gateway a une limitation par défaut de 10 000 requêtes par seconde, alors que Lambda a une limitation par défaut de 1 000. En raison de cette incompatibilité, il est possible qu'API Gateway envoie plus de demandes que ce que Lambda peut traiter. Ce problème peut être résolu en demandant que la limite de simultanéité de Lambda soit augmentée afin qu'elle corresponde au niveau de trafic attendu.

Le test de charge de votre application vous permet de surveiller les performances de votre application end-to-end avant de la déployer en production. Lors d'un test de charge, vous pouvez identifier les quotas susceptibles de limiter les niveaux de trafic que vous attendez et prendre les mesures nécessaires en conséquence.

Historique du document

Le tableau suivant décrit les modifications importantes apportées au Guide du développeur d'AWS Lambda après mai 2018. Pour recevoir les notifications des mises à jour de cette documentation, abonnez-vous au flux RSS.

Modification	Description	Date
AWS mises à jour des politiques gérées	Lambda a publié une nouvelle politique gérée `AWSLambdaBasicDurableExecutionRolePolicy` qui fournit des autorisations d'écriture aux CloudWatch journaux et des autorisations d'exécution durable APIs utilisées par read/write les fonctions durables de Lambda.	1er décembre 2025
Rôle lié à un service pour Lambda	Lambda prend désormais en charge un rôle lié au service () `AWSServiceRoleForLambda` qui permet à Lambda de mettre fin aux instances gérées dans le cadre des fournisseurs de capacité Lambda.	30 novembre 2025
Instances gérées Lambda	Lambda prend désormais en charge les instances gérées, qui fournissent une capacité de calcul dédiée à vos fonctions avec des performances, une sécurité et un contrôle améliorés. Pour plus de détails, consultez	30 novembre 2025

	la section Instances gérées Lambda.	
AWS mises à jour des politiques gérées	Lambda a ajouté une nouvelle politique AWS gérée ()`AWSLambdaBasicDurableExecutionRolePolicy` .	30 novembre 2025
AWS mises à jour des politiques gérées	Lambda a mis à jour (la politique AWS gérée existante `AWSLambda_FullAccess`), autorise les actions `kms:DescribeKey` et`iam:CreateServiceLinkedRole` .	30 novembre 2025
AWS mise à jour des politiques gérée	Lambda a ajouté une nouvelle politique AWS gérée (`AWSLambdaManagedEC2ResourceOperator`) pour permettre la gestion automatisée des EC2 instances Amazon pour les fournisseurs de capacité Lambda. Pour plus de détails, consultez la section Mises à jour Lambda des politiques AWS gérées.	30 novembre 2025
AWS mise à jour des politiques gérée	Lambda a ajouté une nouvelle politique AWS gérée ()`AWSLambdaServiceRolePolicy` . Pour plus de détails, consultez la section Mises à jour Lambda des politiques AWS gérées.	30 novembre 2025

AWS mises à jour des politiques gérées	Lambda a mis à jour deux politiques AWS gérées existantes (`AWSLambda_ReadOnlyAccess` et`AWSLambda_FullAccess`).	20 février 2025
Environnement d'exécution Node.js 22	Lambda prend désormais en charge Node.js 22 en tant qu'environnement d'exécution géré et image de base de conteneur (`nodejs22.x`).	22 novembre 2024
SnapStart support pour Python et .NET	Lambda SnapStart est désormais disponible pour les environnements d'exécution gérés par Python et .NET, en commençant par et. `python3.12` `dotnet8`	18 novembre 2024
Environnement d'exécution Python 3.13	Lambda prend désormais en charge Python 3.13 en tant qu'environnement d'exécution géré et image de base de conteneur.	13 novembre 2024
Chiffrement géré par le client pour les packages de déploiement .zip	Lambda prend désormais en charge le chiffrement des clés géré par le AWS KMS client pour les packages de déploiement .zip.	8 novembre 2024

Support pour SnapStart les nouvelles régions	Lambda SnapStartest désormais disponible dans les régions suivantes : Europe (Espagne), Europe (Zurich), Asie-Pacifique (Melbourne), Asie-Pacifique (Hyderabad) et Moyen-Orient (Émirats arabes unis).	12 janvier 2024
AWS mises à jour des politiques gérées	Lambda a mis à jour une politique AWS gérée existante ()`AWSLambdaVPCAccess ExecutionRole` .	5 janvier 2024
Exécution Python 3.12	Lambda prend désormais en charge Python 3.12 en tant qu'exécution managée et image de base de conteneur . Pour plus d'informations, consultez le runtime Python 3.12 désormais disponible AWS Lambda sur le AWS Compute Blog.	14 décembre 2023
Environnement d’exécution Java 21	Lambda prend désormais en charge Java 21 en tant qu’exécution managée et image de base de conteneur (`java21`).	16 novembre 2023

Environnement d’exécution Node.js 20	Lambda prend désormais en charge Node.js 20 en tant qu’exécution managée et image de base de conteneur (`nodejs20.x`). Pour plus d'informations, consultez le runtime Node.js 20.x désormais disponible AWS Lambda sur le AWS Compute Blog.	14 novembre 2023
Exécution provided.al2023	Lambda prend désormais en charge Amazon Linux 2023 en tant qu’environnement d’exécution géré et image de base de conteneur. Pour plus d'informations, consultez Présentation de l'environnement d'exécution Amazon Linux 2023 AWS Lambda sur le blog AWS Compute.	9 novembre 2023
IPv6 prise en charge des sous-réseaux à double pile	Lambda prend désormais en charge le IPv6 trafic sortant vers des sous-réseaux à double pile. Pour plus d'informations, consultez le IPv6 support.	12 octobre 2023

Test de fonctions et d'applications sans serveur	Découvrez les techniques de débogage et d'automatisation des tests des fonctions sans serveur dans le cloud. Un chapitre sur les tests et des ressources sont désormais inclus dans les sections consacrées aux langages Python et Typescript. Pour en savoir plus, veuillez consulter la rubrique Tester des fonctions et des applications sans serveur.	16 juin 2023
Exécution Ruby 3.2	Lambda prend désormais en charge une nouvelle exécution pour Ruby 3.2. Pour plus d'informations, veuillez consulter la rubrique Création de fonctions Lambda avec Ruby.	7 juin 2023
Streaming des réponses	Lambda prend désormais en charge le streaming des réponses des fonctions. Pour plus d'informations, consultez Configuration d'une fonction Lambda pour le streaming des réponses (français non garanti).	6 avril 2023
Métriques d'invocations asynchrones	Lambda publie des métriques d'invocations asynchrones. Pour plus d'informations, consultez Métriques d'invocations asynchrones.	9 février 2023

Contrôles de version de l'environnement d'exécution	Lambda publie de nouvelles versions de l'environnement d'exécution qui incluent des mises à jour de sécurité, des corrections de bogues et de nouvelles fonctions . Vous pouvez désormais contrôler quand vos fonctions sont mises à jour vers les nouvelles versions de l'environnement d'exécution. Pour plus d'informations, consultez Mises à jour de l'environnement d'exécution Lambda (français non garanti).	23 janvier 2023
Lambda SnapStart	Utilisez Lambda SnapStart pour réduire le temps de démarrage des fonctions Java sans fournir de ressources supplémentaires ni implémenter des optimisations de performances complexes . Pour plus d'informations, consultez la section Amélioration des performances de démarrage avec Lambda SnapStart.	28 novembre 2022

Environnement d’exécution Node.js 18	Lambda prend désormais en charge un nouvel environnement d’exécution (runtime) pour Node.js 18. Node.js 18 utilise Amazon Linux 2. Pour plus d’informations, consultez Création de fonctions Lambda avec Node.js.	18 novembre 2022
lambda : clé de SourceFunctionArn condition	Pour une AWS ressource, la clé de `lambda:SourceFunctionArn` condition filtre l'accès à la ressource par l'ARN d'une fonction Lambda. Pour plus d’informations, consultez utilisation des identifiants d’un environnement d’exécution Lambda.	1et juillet 2022
Environnement d’exécution Node.js 16	Lambda prend désormais en charge un nouvel environnement d’exécution (runtime) pour Node.js 16. Node.js 16 utilise Amazon Linux 2. Pour plus d’informations, consultez Création de fonctions Lambda avec Node.js.	11 mai 2022
Fonction Lambda URLs	Lambda prend désormais en charge les fonctions URLs, qui sont des points de terminaison HTTP (S) dédiés aux fonctions Lambda. Pour plus de détails, consultez la section Fonction Lambda. URLs	6 avril 2022

Événements de test partagés dans la AWS Lambda console	Lambda prend désormais en charge le partage d'événements de test avec d'autres utilisateurs dans le même Compte AWS. Pour plus de détails, consultez Test des fonctions Lambda dans la console.	16 mars 2022
PrincipalOrgId dans les politiques basées sur les ressources	Lambda prend désormais en charge l'octroi d'autorisations à une organisation dans AWS Organizations. Pour plus d'informations, consultez Utilisation de stratégies basées sur les ressources pour AWS Lambda.	11 mars 2022
Exécution .NET 6	Lambda prend désormais en charge un nouvel environnement d'exécution pour .NET 6. Pour en savoir plus, consultez Environnements d'exécution Lambda.	23 février 2022
Filtrage des événements pour les sources d'événements Kinesis, DynamoDB et Amazon SQS	Lambda prend désormais en charge le filtrage des événements pour les sources d'événements Kinesis, DynamoDB et Amazon SQS. Pour plus de détails, consultez Filtrage des événements Lambda.	24 novembre 2021

Authentification MTL pour Amazon MSK et sources d'événements Apache Kafka autogérées	Lambda prend désormais en charge l'authentification MTL pour Amazon MSK et les sources d'événements Apache Kafka autogérées. Pour plus de détails, consultez Utilisation de Lambda avec Amazon MSK.	19 novembre 2021
Lambda sur Graviton2	Lambda prend désormais en charge Graviton2 pour les fonctions utilisant l'architecture arm64. Pour plus de détails, consultez Architectures de l'ensemble des instructions Lambda.	29 septembre 2021
Exécution Python 3.9	Lambda prend désormais en charge un nouveau runtime pour Python 3.9. Pour en savoir plus, consultez Environnements d'exécution Lambda.	16 août 2021
Nouvelles versions d'environnements d'exécution pour Node.js, Python et Java	De nouvelles versions d'environnements d'exécution sont disponibles pour Node.js, Python et Java. Pour en savoir plus, consultez Environnements d'exécution Lambda.	21 juillet 2021

Prise en charge de RabbitMQ en tant que source d'événements sur Lambda	Lambda prend désormais en charge Amazon MQ pour RabbitMQ en tant que source d'événements. Amazon MQ est un service d'agent de messages géré pour Apache ActiveMQ, et RabbitMQ, qui facilite la configuration et l'utilisation d'agents de messages dans le cloud. Pour plus de détails, consultez Utilisation de Lambda avec Amazon MQ.	7 juillet 2021
Authentification SASL/PLAIN pour Kafka autogéré sur Lambda	SASL/PLAIN is now a supported authentication mechanism for self-managed Kafka event sources on Lambda Customers already using SASL/PLAINsur leur cluster Kafka autogéré peuvent désormais facilement utiliser Lambda pour créer des applications grand public sans avoir à modifier leur mode d'authentification. Pour plus d'informations, consultez Utilisation de Lambda avec Apache Kafka autogéré.	29 juin 2021

API d’extensions Lambda	Disponibilité générale pour les extensions Lambda. Utilisez des extensions pour augmenter vos fonctions Lambda. Vous pouvez utiliser des extensions fournies par des partenaires Lambda ou créer vos propres extensions Lambda. Pour plus de détails, consultez API Extensions Lambda.	24 mai 2021
Nouvelle expérience de la console Lambda	La console Lambda a été remaniée pour améliorer les performances et la cohérence.	2 mars 2021
Environnement d’exécution Node.js 14	Lambda prend désormais en charge un nouvel environnement d’exécution (runtime) pour Node.js 14. Node.js 14 utilise Amazon Linux 2. Pour plus d’informations, consultez Création de fonctions Lambda avec Node.js.	27 janvier 2021
Images de conteneurs Lambda	Lambda prend désormais en charge les fonctions définies en tant qu’images conteneurs. Vous pouvez combiner la flexibilité des outils de conteneur avec l’agilité et la simplicité opérationnelle de Lambda pour générer des applications. Pour plus de détails, consultez Utilisation d’images de conteneurs avec Lambda.	1er décembre 2020

Signature de code pour fonctions Lambda	Lambda prend désormais en charge la signature de code. Les administrateurs peuvent configurer des fonctions Lambda de façon à n'accepter que du code signé lors du déploiement. Lambda vérifie les signature s afin de s'assurer que le code n'est pas modifié ou altéré. En outre, Lambda s'assure que le code est signé par des développeurs de confiance avant d'accepter le déploiement. Pour plus de détails, consultez Configura tion de la signature de code pour Lambda.	23 novembre 2020
Aperçu : API de journaux d'exécution Lambda	Lambda prend désormais en charge l'API Runtime Logs. Les extensions Lambda peuvent utiliser l'API Logs pour s'abonner à des flux de journaux dans l'environ nement d'exécution. Pour plus de détails, consultez API Runtime Logs Lambda.	12 novembre 2020

Nouvelle source d'événements pour Amazon MQ	Lambda prend désormais en charge Amazon MQ en tant que source d'événements. Utilisez une fonction Lambda pour traiter les enregistr ements de votre agent de messages Amazon MQ. Pour plus de détails, consultez Utilisation de Lambda avec Amazon MQ.	5 novembre 2020
Aperçu : API d'extensions Lambda	Utilisez des extensions Lambda pour augmenter vos fonctions Lambda. Vous pouvez utiliser des extension s fournies par des partenaires Lambda ou créer vos propres extensions Lambda. Pour plus de détails, consultez API Extensions Lambda.	8 octobre 2020
Support pour Java 8 et environnements d'exécution personnalisés sur AL2	Lambda prend désormais en charge Java 8 et les environne ments d'exécution (runtimes) personnalisés sur Amazon Linux 2. Pour en savoir plus, consultez Environnements d'exécution Lambda.	12 août 2020

Nouvelle source d'événements pour Amazon Managed Streaming for Apache Kafka	Lambda prend désormais en charge Amazon MSK en tant que source d'événements. Utilisez une fonction Lambda avec Amazon MSK pour traiter des enregistrements dans une rubrique Kafka. Pour plus de détails, consultez Utilisation de Lambda avec Amazon MSK.	11 août 2020
Clés de condition IAM pour les paramètres du VPC Amazon	Vous pouvez désormais utiliser des clés de condition spécifiques de Lambda pour les paramètres du VPC. Par exemple, vous pouvez exiger que toutes les fonctions de votre organisation soient connectées à un VPC. Vous pouvez également spécifier les sous-réseaux et les groupes de sécurité que les utilisateurs de la fonction peuvent et ne peuvent pas utiliser. Pour plus de détails, consultez Configuration du VPC pour les fonctions IAM.	10 août 2020

Paramètres de simultanéité pour les consommateurs de flux Kinesis HTTP/2	Vous pouvez désormais utiliser les paramètres de simultanéité suivants pour les utilisateurs de Kinesis dotés d'un système de ventilation amélioré (flux HTTP/2) ParallelizationFactor : MaximumRetryAttempts,, et. MaximumRecordAgeInSeconds DestinationConfig BisectBatchOnFunctionError Pour plus de détails, consultez la section Utilisation AWS Lambda avec Amazon Kinesis.	7 juillet 2020
Fenêtre de traitement par lots pour les utilisateurs de flux Kinesis HTTP/2	Vous pouvez désormais configurer une fenêtre de traitement par lots (MaximumBatchingWindowInSeconds) pour les flux HTTP/2. Lambda lit les enregistrements du flux jusqu'à avoir rassemblé un lot complet, ou jusqu'à l'expiration de la fenêtre de traitement par lots. Pour plus de détails, consultez la section Utilisation AWS Lambda avec Amazon Kinesis.	18 juin 2020

Prise en charge des systèmes de fichiers Amazon EFS	Vous pouvez désormais connecter un système de fichiers Amazon EFS à vos fonctions Lambda pour un accès réseau partagé aux fichiers. Pour plus de détails, consultez Configuration de l'accès au système de fichiers pour les fonctions Lambda.	16 juin 2020
AWS CDK exemples d'applications dans la console Lambda	La console Lambda inclut désormais des exemples d'applications utilisant le AWS Cloud Development Kit (AWS CDK) for. TypeScript AWS CDK Il s'agit d'un framework qui vous permet de définir les ressources de votre application en Python TypeScript, Java ou .NET.	1 juin 2020
Support de l'environnement d'exécution .NET Core 3.1.0 dans AWS Lambda	AWS Lambda prend désormais en charge le runtime .NET Core 3.1.0. Pour en savoir plus, consultez la page .NET Core CLI.	31 mars 2020

Support pour API Gateway HTTP APIs	Documentation mise à jour et étendue sur l'utilisation de Lambda avec API Gateway, y compris la prise en charge du protocole HTTP. APIs Ajout d'un exemple d'application qui crée une API et une fonction avec CloudFormation. Pour plus de détails, consultez Utilisation de Lambda avec Amazon API Gateway.	23 mars 2020
Ruby 2.7	Un nouvel environnement d’exécution est disponible pour Ruby 2.7, ruby2.7, qui est le premier environnement d’exécution Ruby à utiliser Amazon Linux 2. Pour plus d’informations, consultez Création de fonctions Lambda avec Ruby.	19 février 2020

Métriques de simultanéité	Lambda rapporte désormais les métriques `ConcurrentExecutions` pour l'ensemble des fonctions, alias et versions. Vous pouvez afficher un graphique pour cette métrique dans la page de surveillance de votre fonction. Auparavant, `ConcurrentExecutions` était seulement signalé au niveau du compte et pour les fonctions utilisant la simultanéité réservée. Pour plus d'informations, consultez AWS Lambda Métriques des fonctions.	18 février 2020

Mise à jour des états des fonctions	Les états de fonction sont désormais appliqués pour toutes les fonctions par défaut. Lorsque vous connectez une fonction à un VPC, Lambda crée des interfaces réseau Elastic partagées. Cela permet à votre fonction d'évoluer sans créer d'interfaces réseau supplémentaires. Pendant ce temps, vous ne pouvez pas effectuer d'opérations supplémentaires sur la fonction, y compris la mise à jour de sa configuration et de ses versions de publication. Dans certains cas, l'invocation est également affectée. Les détails sur l'état actuel d'une fonction sont disponibles à partir de l'API Lambda. Cette mise à jour est publiée en plusieurs phases. Pour plus de détails, consultez la section Mise à jour du cycle de vie des états Lambda pour les réseaux VPC sur le blog Compute. AWS Pour plus d'informations sur les états, consultez États de fonctions AWS Lambda.	24 janvier 2020

Mises à jour de la sortie de l'API de configuration de fonction	Ajout de codes de motif à StateReasonCode(InvalidSubnet, InvalidSecurityGroup) et LastUpdateStatusReasonCode (SubnetOutOfIPAddresses, InvalidSubnet, InvalidSecurityGroup) pour les fonctions connectées à un VPC. Pour plus d'informations sur les états, consultez États de fonctions AWS Lambda.	20 janvier 2020
Simultanéité allouée	Vous pouvez désormais allouer la simultanéité provisionnée pour une version de fonction ou un alias. La simultanéité provisionnée permet à une fonction d'évoluer sans fluctuations de latence. Pour plus de détails, consultez Gestion de la simultanéité pour une fonction Lambda.	3 décembre 2019

Créer un proxy de base de données	Vous pouvez désormais utiliser la console Lambda pour créer un proxy de base de données pour une fonction Lambda. Un proxy de base de données permet à une fonction d'atteindre des niveaux de simultanéité élevés sans épuiser les connexion s de base de données. Pour plus de détails, consultez Configuration de l'accès à une base de données pour une fonction Lambda.	3 décembre 2019
Prise en charge des percentil es pour la métrique de durée	Vous pouvez désormais filtrer la métrique de durée en fonction des percentiles. Pour plus d'informations, consultez Métriques AWS Lambda.	26 novembre 2019
Augmentation de la simultané ité pour des sources d'événement de flux	Une nouvelle option pour les mappages de source d'événements de flux DynamoDB et de flux Kinesis vous permet de traiter plusieurs lots à la fois à partir de chaque partition. Lorsque vous augmentez le nombre de lots simultanés par partition , la simultanéité de votre fonction peut atteindre dix fois le nombre de partitions dans votre flux. Pour en savoir plus, consultez Mappage de source d'événement Lambda.	25 novembre 2019

États des fonctions	Lorsque vous créez ou mettez à jour une fonction, celle-ci passe en état d'attente pendant que Lambda approvisionne les ressources pour sa prise en charge. Si vous connectez votre fonction à un VPC, Lambda peut créer immédiatement une interface réseau Elastic partagée, au lieu de créer des interfaces réseau lors de l'invocation de votre fonction. Cela se traduit par de meilleures performances pour les fonctions connectées au VPC, mais peut nécessiter une mise à jour de votre automatisation. Pour plus d'informations, consultez États de fonction AWS Lambda.	25 novembre 2019
Options de gestion des erreurs pour les invocations asynchrones	De nouvelles options de configuration sont disponibles pour l'invocation asynchrone. Vous pouvez configurer Lambda de manière à limiter les nouvelles tentatives et à définir un âge d'événement maximal. Pour plus de détails, consultez Configuration de la gestion des erreurs pour les invocations asynchrones.	25 novembre 2019

Gestion des erreurs pour les sources d'événement de flux	De nouvelles options de configuration sont disponibles pour les mappages de source d'événement qui lisent depuis les flux. Vous pouvez configurer des mappages de source d'événements DynamoDB Streams et Kinesis Streams pour limiter les nouvelles tentatives et définir une ancienneté de registre maximale. Lorsque des erreurs se produisent, vous pouvez configurer le mappage de source d'événement pour fractionner les lots avant d'effectuer des nouvelles tentatives et d'envoyer des enregistrements d'invocations pour les lots ayant échoué vers une file d'attente ou une rubrique. Pour en savoir plus, consultez Mappage de source d'événement Lambda.	25 novembre 2019

Destinations pour les invocations asynchrones	Vous pouvez désormais configurer Lambda pour envoyer des enregistrements d'invocations asynchrones à un autre service. Les enregistrements d'invocation contiennent des détails sur l'événement, le contexte et la réponse de la fonction. Vous pouvez envoyer des enregistrements d'invocation vers une file d'attente SQS, une rubrique SNS, une fonction Lambda ou un bus d'événements. EventBridge Pour plus de détails, consultez Configuration des destinations pour les invocations asynchrones.	25 novembre 2019
Nouveaux environnements d'exécution pour Node.js, Python et Java	De nouveaux environnements d'exécution sont disponibles pour Node.js 12, Python 3.8 et Java 11. Pour en savoir plus, consultez Environnements d'exécution Lambda.	18 novembre 2019
Prise en charge des files d'attente FIFO pour les sources d'événement Amazon SQS	Vous pouvez désormais créer un mappage de source d'événement qui lit à partir d'une file d'attente FIFO (premier entré, premier sorti). Auparavant, seules les files d'attente standard étaient prises en charge. Pour plus de détails, consultez Utilisation de Lambda avec Amazon SQS.	18 novembre 2019

Créer des applications dans la console Lambda	La création d'applications dans la console Lambda est désormais généralement disponible. Pour obtenir des instructions, consultez Managing applications in the Lambda console.	31 octobre 2019
Créer des applications dans la console Lambda (bêta)	Vous pouvez désormais créer une application Lambda avec un pipeline de livraison continue intégré dans la console Lambda. La console fournit des exemples d'applications que vous pouvez utiliser comme point de départ pour votre propre projet. Choisissez entre AWS CodeCommit et GitHub pour le contrôle de source. Chaque fois que vous transmettez des modifications à votre référentiel, le pipeline inclus les construit et les déploie automatiquement. Pour obtenir des instructions, consultez Managing applications in the Lambda console.	3 octobre 2019

Améliorations des performances pour les fonctions connectées à un VPC	Lambda utilise désormais un nouveau type d'interface réseau Elastic qui est partagé par toutes les fonctions dans un sous-réseau Virtual Private Cloud (VPC). Lorsque vous connectez une fonction à un VPC, Lambda crée une interface réseau pour chaque combinaison de groupe de sécurité et de sous-rése au que vous choisissez. Lorsque les interfaces réseau partagées sont disponibles, la fonction n'a plus besoin de créer d'interfaces réseau supplémentaires au fur et à mesure de sa mise à l'échelle ascendante. Ainsi, les temps de démarrage sont considéra blement améliorés. Pour plus d'informations, consultez Configuration d'une fonction Lambda pour accéder aux ressources d'un VPC.	3 septembre 2019

Paramètres dpour les lots de flux	Vous pouvez désormais configurer une fenêtre de traitement par lots pour les mappages de source d'événements Amazon DynamoDB et Amazon Kinesis. Configurez une fenêtre de traitement par lots de cinq minutes maximum pour mettre en mémoire tampon les enregistrements entrants jusqu'à ce qu'un lot complet soit disponible. Cela réduit le nombre de fois où votre fonction est invoquée lorsque le flux est moins actif.	29 août 2019
CloudWatch Intégration de Logs Insights	La page de surveillance de la console Lambda inclut désormais les rapports d'Amazon CloudWatch Logs Insights.	18 juin 2019
Amazon Linux 2018.03	L'environnement d'exécution Lambda est en cours de mise à jour pour utiliser Amazon Linux 2018.03. Pour plus de détails, consultez Environnement d'exécution.	21 mai 2019

Node.js 10	Un nouvel environnement d'exécution est disponible pour Node.js 10, nodejs10.x. Cet environnement d'exécution utilise Node.js 10.15 et sera mis à jour avec la dernière version de Node.js 10 régulièrement. Node.js 10 est également le premier environnement d'exécution pour utiliser Amazon Linux 2. Pour plus d'informations, consultez Création de fonctions Lambda avec Node.js.	13 mai 2019
GetLayerVersionByArn API	Utilisez l'GetLayerVersionByArnAPI pour télécharger les informations de version de la couche avec l'ARN de version en entrée. Par rapport à GetLayerVersion, vous GetLayerVersionByArn permet d'utiliser directement l'ARN au lieu de l'analyser pour obtenir le nom de la couche et le numéro de version.	25 avril 2019
Ruby	AWS Lambda supporte désormais Ruby 2.5 avec un nouveau runtime. Pour plus d'informations, consultez Création de fonctions Lambda avec Ruby.	29 novembre 2018

Couches	Avec les couches Lambda, vous pouvez regrouper et déployer des bibliothèques, des environnements d'exécution personnalisés, ainsi que d'autres dépendances séparément depuis le code de votre fonction. Partagez vos couches avec vos autres comptes ou avec le monde entier. Pour plus de détails, consultez Couches Lambda.	29 novembre 2018
Exécutions personnalisées	Créez un environnement d'exécution (runtime) personnalisé pour exécuter des fonctions Lambda dans votre langage de programmation favori. Pour plus de détails, consultez Runtimes Lambda personnalisés.	29 novembre 2018
Déclencheurs d'Application Load Balancer	Elastic Load Balancing prend désormais en charge les fonctions Lambda en tant que cibles pour les équilibreurs de charge d'applications. Pour plus de détails, consultez Utilisation de Lambda avec des équilibreurs de charge des applications.	29 novembre 2018

Utilisation des consommateurs de flux Kinesis HTTP/2 comme déclencheur	Vous pouvez utiliser les consommateurs de flux de données Kinesis HTTP/2 pour envoyer des événement s à AWS Lambda. Les consommateurs de flux dédiés ont un débit de lecture dédié à partir de chaque partition dans votre flux de données et utilisent HTTP/2 pour réduire la latence. Pour plus de détails, consultez Utilisation de Lambda avec Kinesis.	19 novembre 2018
Python 3.7	AWS Lambda supporte désormais Python 3.7 avec un nouveau runtime. Pour plus d'informations, consultez Création de fonctions Lambda avec Python.	19 novembre 2018
Augmentation de la limite de la charge utile d'invocation de fonction asynchrone	La taille maximum de charge utile pour les invocations asynchrones a augmenté de 128 Ko à 256 Ko, et correspon d à la taille maximum de message émis par un déclencheur Amazon SNS. Pour plus de détails, consultez Quotas Lambda.	16 novembre 2018
AWS GovCloud Région (USA Est)	AWS Lambda est désormais disponible dans la région AWS GovCloud (USA Est).	12 novembre 2018

AWS SAM Rubriques déplacées vers un guide du développeur distinct	Un certain nombre de sujets étaient axés sur la création d'applications sans serveur à l'aide de AWS Serverles s Application Model (AWS SAM). Ces rubriques ont été déplacées vers le Guide du développeur AWS Serverless Application Model.	25 octobre 2018
Afficher des applications Lambda dans la console	Vous pouvez afficher l'état de vos applications Lambda sur la page Applications dans la console Lambda. Cette page indique l'état de la CloudForm ation pile. Elle inclut des liens vers les pages où vous pouvez consulter plus d'informations sur les ressources figurant dans la pile. Vous pouvez également consulter les métriques agrégées pour l'application et créer des tableaux de bord de surveilla nce personnalisés.	11 octobre 2018
Délai d'attente d'exécution de la fonction	Pour permettre d'utiliser des fonctions de longue durée, le délai d'exécution maximal configurable est passé de 5 minutes à 15 minutes. Pour plus de détails, consultez Limites Lambda.	10 octobre 2018

Support du langage PowerShell de base dans AWS Lambda	AWS Lambda supporte désormais le langage PowerShell de base. Pour plus d'informations, voir Modèle de programmation pour la création de fonctions Lambda dans. PowerShell	11 septembre 2018
Support de l'environnement d'exécution .NET Core 2.1.0 dans AWS Lambda	AWS Lambda prend désormais en charge le runtime .NET Core 2.1.0. Pour en savoir plus, veuillez consulter la page .NET Core CLI.	9 juillet 2018
Mises à jour disponibles sur RSS	Vous pouvez maintenant vous abonner à un flux RSS pour suivre les versions de ce guide.	5 juillet 2018
Prise en charge d'Amazon SQS en tant que source d'événements	AWS Lambda prend désormais en charge Amazon Simple Queue Service (Amazon SQS) en tant que source d'événements. Pour plus d'informations, consultez Invocation des fonctions Lambda.	28 juin 2018

Région Chine (Ningxia)	AWS Lambda est désormais disponible dans la région Chine (Ningxia). Pour plus d'informations sur les régions et les points de terminaison Lambda, consultez Régions et points de terminaison (français non garanti) dans la Références générales AWS.	28 juin 2018

Mises à jour antérieures

Le tableau ci-après décrit les modifications importantes apportées dans chaque version du Manuel du développeur AWS Lambda avant juin 2018.

Modification	Description	Date
Prise en charge de l'environnement d'exécution Node.js 8.10	AWS Lambda prend désormais en charge la version 8.10 du runtime Node.js. Pour de plus amples informations, veuillez consulter Création de fonctions Lambda avec Node.js.	2 avril 2018
Révision des fonctions et des alias IDs	AWS Lambda prend désormais en charge la révision IDs des versions et des alias de vos fonctions. Vous pouvez les utiliser IDs pour suivre et appliquer des mises à jour conditionnelles lorsque vous mettez à jour la version de votre fonction ou vos ressources d'alias.	25 janvier 2018
Prise en charge de l'environnement d'exécution Go et .NET 2.0	AWS Lambda a ajouté le support d'exécution pour Go et .NET 2.0. Pour plus d'informations, consultez Création de fonctions Lambda avec Go et Création de fonctions Lambda avec C#.	15 janvier 2018
Nouvelle conception de la console	AWS Lambda a introduit une nouvelle console Lambda pour simplifier votre expérience et a ajouté un éditeur de	30 novembre 2017

Modification	Description	Date
	code Cloud9 pour améliorer votre capacité à déboguer et à réviser votre code de fonction.	
Définition de limites de simultanéité pour des fonctions individuelles	AWS Lambda permet désormais de définir des limites de simultanéité pour des fonctions individuelles. Pour de plus amples informations, veuillez consulter Configuration de la simultanéité réservée pour une fonction.	30 novembre 2017
Déplacement du trafic avec des alias	AWS Lambda prend désormais en charge le transfert de trafic avec des alias. Pour de plus amples informations, veuillez consulter Création d'un déploiement continu avec des alias pondérés.	28 novembre 2017
Déploiement de code graduel	AWS Lambda permet désormais de déployer en toute sécurité de nouvelles versions de votre fonction Lambda en tirant parti de Code Deploy. Pour plus d'informations, consultez Déploiement de code graduel.	28 novembre 2017
Région Chine (Pékin)	AWS Lambda est désormais disponible dans la région Chine (Pékin). Pour plus d'informations sur les régions et les points de terminaison Lambda, consultez Régions et points de terminaison (français non garanti) dans la Références générales AWS.	9 novembre 2017
Présentation de SAM Local	AWS Lambda présente SAM Local (désormais connu sous le nom de SAM CLI), un AWS CLI outil qui fournit un environnement dans lequel vous pouvez développer, tester et analyser vos applications sans serveur localemen t avant de les télécharger sur le runtime Lambda. Pour plus d'informations, consultez Test et débogage d'applications sans serveur.	11 août 2017

Modification	Description	Date
Canada (Central) Region	AWS Lambda est désormais disponible dans la région du Canada (Centre). Pour plus d'informations sur les régions et les points de terminaison Lambda, consultez Régions et points de terminaison (français non garanti) dans la Références générales AWS.	22 juin 2017
South America (São Paulo) Region	AWS Lambda est désormais disponible dans la région Amérique du Sud (São Paulo). Pour plus d'informations sur les régions et les points de terminaison Lambda, consultez Régions et points de terminaison (français non garanti) dans la Références générales AWS.	6 juin 2017
AWS Lambda support pour AWS X-Ray.	Lambda prend désormais en charge X-Ray, ce qui vous permet de détecter, d'analyser et d'optimiser les problèmes de performance avec les applications Lambda. Pour plus d'informations, consultez Visualisez les invocations de fonctions Lambda à l'aide de AWS X-Ray.	19 avril 2017
Asia Pacific (Mumbai) Region	AWS Lambda est désormais disponible dans la région Asie-Pacifique (Mumbai). Pour plus d'informations sur les régions et les points de terminaison Lambda, consultez Régions et points de terminaison (français non garanti) dans la Références générales AWS.	28 mars 2017
AWS Lambda prend désormais en charge le runtime Node.js v6.10	AWS Lambda ajout du support pour le runtime Node.js v6.10. Pour de plus amples informations, veuillez consulter Création de fonctions Lambda avec Node.js.	22 mars 2017
Europe (London) Region	AWS Lambda est désormais disponible dans la région Europe (Londres). Pour plus d'informations sur les régions et les points de terminaison Lambda, consultez Régions et points de terminaison (français non garanti) dans la Références générales AWS.	1 février 2017

Modification	Description	Date
AWS Lambda prise en charge de l'environnement d'exécution .NET, de Lambda @Edge (version préliminaire), de Dead Letter Queues et du déploiement automatisé d'applications sans serveur.	AWS Lambda ajout du support pour C#. Pour de plus amples informations, veuillez consulter Création de fonctions Lambda avec C#. Lambda @Edge vous permet d'exécuter des fonctions Lambda sur les sites AWS Edge en réponse à des événements. CloudFront Pour plus d'informations, consultez Customize at the edge with Lambda@Edge.	3 décembre 2016
AWS Lambda ajoute Amazon Lex en tant que source d'événements prise en charge.	Avec Lambda et Amazon Lex, vous pouvez créer rapidement des chatbots pour divers services, comme Slack et Facebook.	30 novembre 2016
US West (N. California) Region	AWS Lambda est désormais disponible dans la région USA Ouest (Californie du Nord). Pour plus d'informations sur les régions et les points de terminaison Lambda, consultez Régions et points de terminaison (français non garanti) dans la Références générales AWS.	21 novembre 2016

Modification	Description	Date
Introduction de la fonctionnalité AWS SAM permettant de créer et de déployer des applications basées sur Lambda et d'utiliser des variables d'environnement pour les paramètres de configuration des fonctions Lambda.	AWS SAM: vous pouvez désormais utiliser le AWS SAM pour définir la syntaxe d'expression des ressources dans une application sans serveur. Pour déployer votre applicati on, il vous suffit de spécifier les ressources dont vous avez besoin dans le cadre de cette dernière, ainsi que les stratégies d'autorisations qui leur sont associées, dans un fichier de modèle AWS CloudFormation (au format JSON ou YAML), d'empaqueter vos artefacts de déploiement et de déployer le modèle. Variables d'environnement : vous pouvez utiliser des variables d'environnement afin de spécifier des paramètre s de configuration pour votre fonction Lambda en dehors du code de votre fonction. Pour plus d'informations, consultez Utilisation des variables d'environnement Lambda.	18 novembre 2016
Asia Pacific (Seoul) Region	AWS Lambda est désormais disponible dans la région Asie-Pacifique (Séoul). Pour plus d'informations sur les régions et les points de terminaison Lambda, consultez Régions et points de terminaison (français non garanti) dans la Références générales AWS.	29 août 2016
Asia Pacific (Sydney) Region	Lambda est désormais disponible dans la région Asie-Paci fique (Sydney). Pour plus d'informations sur les régions et les points de terminaison Lambda, consultez Régions et points de terminaison (français non garanti) dans la Références générales AWS.	23 juin 2016
Mises à jour de la console Lambda	La console Lambda a été mise à jour pour simplifier le processus de création de rôles.	23 juin 2016
AWS Lambda prend désormais en charge le runtime Node.js v4.3	AWS Lambda ajout du support pour le runtime Node.js v4.3. Pour de plus amples informations, veuillez consulter Création de fonctions Lambda avec Node.js.	07 avril 2016

Modification	Description	Date
Région Europe (Frankfurt)	Lambda est désormais disponible dans la région Europe (Frankfurt). Pour plus d'informations sur les régions et les points de terminaison Lambda, consultez Régions et points de terminaison (français non garanti) dans la Références générales AWS.	14 mars 2016
Prise en charge de VPC	Vous pouvez maintenant configurer une fonction Lambda pour accéder aux ressources de votre VPC. Pour plus d'informations, consultez Octroi aux fonctions Lambda d'un accès aux ressources d'un Amazon VPC.	11 février 2016
Le runtime Lambda a été mis à jour.	L'environnement d'exécution a été mis à jour.	4 novembre 2015

Modification	Description	Date
Prise en charge de la gestion des versions, Python pour le développement de code pour des fonctions Lambda, événements planifiés et allongement du temps d'exécution	Vous pouvez désormais développer le code de la fonction Lambda à l'aide du langage Python. Pour plus d'informations, consultez Création de fonctions Lambda avec Python. Gestion des versions : vous pouvez conserver une ou plusieurs versions de la fonction Lambda. La gestion des versions vous permet de contrôler quelle version de fonction Lambda est exécutée dans des environnements différents (par exemple, développement, test ou production). Pour plus d'informations, consultez Gestion des versions d'une fonction Lambda. Événements planifiés : vous pouvez également configurer Lambda pour invoquer régulièrement le code à l'aide de la console Lambda. Vous pouvez spécifier un taux fixe (nombre d'heures, de jours ou de semaines) ou vous pouvez spécifier une expression cron. Pour de plus amples informations, veuillez consulter Invocation d'une fonction Lambda dans une planification. Rallongement du temps d'exécution : vous pouvez maintenant définir un temps d'exécution pouvant allant jusqu'à cinq minutes pour les fonctions Lambda afin de rendre possibles les fonctions de plus longue durée, telles que l'ingestion de données volumineuses et le traitement des tâches.	08 octobre 2015

Modification	Description	Date
Prise en charge de DynamoDB Streams	DynamoDB Streams est désormais disponible dans toutes les régions qui acceptent DynamoDB. Vous pouvez activer DynamoDB Streams pour votre table et utiliser une fonction Lambda comme déclencheur de cette table. Les déclencheurs sont des actions personnalisées que vous effectuez en réponse aux mises à jour apportées à la table DynamoDB. Pour afficher un exemple de procédure, veuillez consulter Tutoriel : Utilisation AWS Lambda avec les flux Amazon DynamoDB .	14 juillet 2015
Lambda prend désormais en charge l'invocation des fonctions Lambda avec des clients compatibles REST.	Jusqu'à présent, pour appeler votre fonction Lambda depuis votre application Web, mobile ou IoT, vous aviez besoin du AWS SDKs (par exemple, un AWS SDK pour Java, un SDK pour AWS Android ou un SDK pour iOS). AWS Désormais, Lambda prend en charge l'invocation d'une fonction Lambda avec des clients compatibles REST via une API personnalisée que vous pouvez créer à l'aide d'Amazon API Gateway. Vous pouvez envoyer les requêtes à l'URL du point de terminaison de la fonction Lambda. Vous pouvez configurer la sécurité au niveau du point de terminaison pour autoriser l'accès ouvert, pour tirer parti d' Gestion des identités et des accès AWS (IAM) et autoriser l'accès ou pour utiliser des clés d'API pour le contrôle de l'accès à vos fonctions Lambda par des tiers. Pour voir un exemple d'exercice de mise en route, veuillez consulter Invocation d'une fonction Lambda à l'aide d'un point de terminaison Amazon API Gateway.	09 juillet 2015

Modification	Description	Date
La console Lambda fournit maintenant des plans pour créer facilement des fonctions Lambda et les tester.	La console Lambda fournit un ensemble de plans. Chaque plan fournit un exemple de configuration de source d'événement et un exemple de code que vous pouvez utiliser pour créer facilement des applications basées sur Lambda. Tous les exercices de mise en route Lambda utilisent désormais les plans.	09 juillet 2015
Lambda prend désormais en charge Java pour créer vos fonctions Lambda.	Vous pouvez désormais créer le code Lambda en Java. Pour plus d'informations, consultez Création de fonctions Lambda avec Java.	15 juin 2015
Lambda prend désormais en charge la spécification d'un objet Amazon S3 en tant que fonction .zip lorsque vous créez ou mettez à jour une fonction Lambda.	Vous pouvez importer un package de déploiement de fonctions Lambda (fichier .zip) dans un compartiment Amazon S3 de la région dans laquelle vous souhaitez créer une fonction Lambda. Vous pouvez ensuite spécifier le nom du compartiment et le nom de la clé objet lorsque vous créez ou mettez à jour une fonction Lambda.	28 mai 2015
Lambda est désormais disponible de manière globale et prend en charge les backends mobiles	Lambda est désormais disponible de manière globale en production. Cette version présente également de nouvelles fonctionnalités qui facilitent la création de backends pour les mobiles, les tablettes et l'Internet des objets (IoT) via Lambda, qui adaptent automatiquement leur échelle, sans que vous ayez à mettre en service ou à gérer une infrastructure. Lambda prend désormais en charge les événements en temps réel (synchrones) et asynchrones. Les fonctionnalités supplémentaires comprennent la rationalisation de la configuration et de la gestion des sources d'événement. Le modèle d'autorisation et le modèle de programmation ont été simplifiés par la mise en place de stratégies de ressources pour les fonctions Lambda.	9 avril 2015

Modification	Description	Date
Version préliminaire	Version préliminaire du Guide du développeur AWS Lambda .	13 novembre 2014

Les traductions sont fournies par des outils de traduction automatique. En cas de conflit entre le contenu d'une traduction et celui de la version originale en anglais, la version anglaise prévaudra.